Chapter 3
The K6 3D Microarchitecture

In this chapter, we will explore three main aspects of the microarchitecture of the K6 3D in more detail: its scheduler, its operation commit unit, and its register renaming scheme. As specific microarchitectural concepts are introduced, pseudo-RTL code is given for typical chunks of logic that could be used to implement these concepts.

Road Map of Chapter 3

Section	Audience
All sections in this chapter	Practitioners, University Professors and Students
Chapter Summary	All

The Scheduler: An Expanded Description

In Chapter 2 we showed that the scheduler:

1. is tightly coupled to the execution units.
2. is tightly coupled to the OCU.
3. maintains information concerning Ops in multiple execution pipelines.
4. issues Ops to the execution units for execution.
5. provides the Op information to the execution units at the times required.
6. does register and status flag dependency checking.
7. forwards results as required for the execution of dependent Ops.
8. holds the results from completed Ops until the results are committed or aborted.
9. deletes Op information as required.

In accomplishing this the scheduler can initiate, among other things:

1. four Op issues, to Load Unit, Store Unit, Register Unit X, and Register Unit Y.
2. nine register operand "fetches": two each for Load Unit, Registrant X, and Register Unit Y and three for the Store Unit, including immediate values for Register Unit X and Register Unit Y.
3. two displacement "fetches," for Load Unit and Store Unit.
4. outputs to nine register operand buses.
5. one status operand fetch, for Register Unit X.
6. BRCOND Op resolution and mispredicted branch handling.
7. LdOp and StOp ordering and relative age determination for pipeline Stage 2 LdOps and StOps.

We will now examine how all of this is done.

Loading the Scheduler

It might be useful at this point for you to review the scheduler diagram given in Figure 2.9 on page 130 and the three diagrams related to the decoder, i.e. Figure 2.6 on page 115, Figure 2.8 on page 127, and Figure 2.10 on page 139.

As OpQuads are loaded into the scheduler, they pass through the OpQuad Expansion Logic discussed in the "Decoder and Scheduler OpQuads" section in Chapter 2 and shown in Figure 2.8 on page 127. The OpQuad Expansion Logic expands the *decoder* OpQuads into *scheduler* OpQuads. *Decoder* OpQuads are 152 bits wide and *scheduler* OpQuads are 578 bits wide. Different types of processing occur during the expansion of *decoder* OpQuads:

1. a few Op fields are simply passed through unchanged.
2. some Op fields are modified based on the values contained in other fields.
3. some Op fields are replaced by physically different fields.
4. new fields are derived from existing ones.

As explained in the section titled "Execution Pipelines" beginning on page 158 and as shown in Figure 2.8 (cited above), Figure 2.6 on page 115 and Figure 2.10 on page 139, OpQuad Templates, fetched from the OpQuad ROM, are translated into *decoder* OpQuads in the environment substitution logic preceding the OpQuad Expansion Logic.

Whatever its source, the *scheduler* OpQuad formed by the OpQuad Expansion Logic is loaded into the scheduler whenever Row 0 (the top row) of the scheduler's buffer is empty or contains an OpQuad that is shifting to Row 1. If no scheduler OpQuad is available when the OpQuad in the top row shifts down, an *invalid* OpQuad is loaded into Row 0.

scheduler entry

In Figure 2.9 on page 130, the scheduler's centralized buffer[22] is represented as consisting of six rows, each row having four entries—one entry for each Op in the scheduler OpQuad stored in that row. Each of these entries includes:

1. a number of static and dynamic fields, i.e., values stored in storage elements.
2. various portions of logic dedicated to specific scheduler functions—e.g., the *issue selection* logic, the *operand selection* logic, the *load/store ordering* logic, and the *self-modifying code support* logic.

static and dynamic fields

The entries in Rows 3, 4, and 5 in Figure 2.9 also contain status flag-dependent RegOp synchronization logic, branch resolution logic, and status flag fetch logic. All twenty-four Op entries are otherwise essentially identical.

Most of each entry's storage elements contain values of static fields. These fields are initialized when the Op is loaded into the scheduler and maintain their value until the Op is retired. Their contents can never change from their initial values. Other storage elements for an entry contain values of dynamic fields which can be reloaded with new values before the Op is retired.

OpQuad fields

Finally, other storage elements store values for fields, referred to as OpQuad fields that are associated with a scheduler OpQuad taken as a whole. Most OpQuad fields are static; however, some OpQuad fields are dynamic.

DEFINITION

Static Fields and Dynamic Fields

Some fields in an Op entry in the scheduler's buffer retain the same value throughout execution of the Op and are called static fields. Other fields can be changed as the Op proceeds through the scheduler and are called dynamic fields.

When taken on an OpQuad basis, the scheduler's storage elements can be thought of as forming a shift register that is six rows deep. Each clock cycle, an OpQuad that is not held up in a row shifts down to the next row if the next row is empty or contains an OpQuad that is also shifting downward. The OpQuad in the bottom row shifts out of the scheduler if all operations associated with the bottom row have been committed or aborted.

[22] Instruction windows, reservation stations, and centralized buffers were introduced in the section titled "Superscalar Design" beginning on page 74.

Historical Comment and Suggested Readings

Early Environment Substitution Techniques

Designers of some early microprogrammable machines recognized that a number of microcode sequences used to emulate instruction set architectures were quite similar, differing, for example, only in the specific registers manipulated. To take advantage of this situation, the specific registers used in the microcode sequence were specified in auxiliary registers whose values were "merged" with the microcode sequence as it was being executed. Such implementations seem to be precursors to both OpQuad Template environment substitution and to the use of dynamic fields. See, for example, "Scheme-79—Lisp on a Chip," by Gerald Jay Sussman, Jack Holloway, Guy Lewis Steel, Jr., and Alan Bell, in *Computer*, July 1981, pp. 10-21.

The entries in the scheduler's buffer store information regarding the Ops that are awaiting execution, being executed, or have completed execution. How and when the information in the storage elements is modified is central to the operation of the scheduler. For example, the value of the State Field of an entry changes as an Op proceeds through the scheduler to indicate if the Op has been issued, is in a specific stage of execution, or has been completed.

Notation

OpQuad (without a modifier)

From now on the term "OpQuad" will be used without the modifier "*decoder*" or "*scheduler*," unless either is needed for clarity. It should be clear from the context if it refers to a *decoder OpQuad* or a *scheduler OpQuad*. In the current section, we are discussing scheduler OpQuads.

The scheduler issues Ops, provides Op information to the execution units at a specific time when required, holds the results from completed Ops until the results are committed or aborted, and forwards the results for the execution of other Ops as required. Each scheduler entry holds the register and status flags results from its associated Op. Each dynamic field storage element must be able to be loaded both from the appropriate preceding storage element and from a relevant data source. Further, the loading of data into a storage element and the shifting of the present or new data into a different storage element must be able to happen simultaneously (during the same cycle) and independently (from a control perspective).

Design Note

Scheduler Entries Hold Register and Status Flags Results

Each scheduler entry holds the register and status flags results produced, if any, by the execution of its associated Op. This is a key element of the K6's implicit register renaming scheme discussed later in this chapter.

Shifting OpFields from Row to Row

Figure 3.1 on page 187 shows portions of the scheduler's buffer that represent parts of an entry in Row 3 and its connection to Rows 2 and 4. Figure 3.1 shows a storage element for one of that entry's static fields while Figure 3.2 on page 191 shows a storage element for one of its dynamic fields. Both storage elements are edge-triggered flip-flops. Row 3 contains identical storage elements for every bit of each dynamic and static field for all four entries in Row 3 (i.e., the four Ops in Row 3). What this figure represents then is one of the four Ops (and the same Op) in each of the three rows. The other rows in the buffer are similar or identical to Row 3 and are connected in series with Row 3.

Let's look into this example in more detail. In Figure 3.1, flip-flops A, B, and C store one bit of the same static field in Rows 2, 3, and 4. This bit value shifts from flip-flop A to flip-flop B to flip-flop C as the OpQuad shifts from Row 2 to Row 3 to Row 4. The global control logic generates signals LdEntry[i], one for each row to control whether shifts to the corresponding rows occur or not. Thus, the LdEntry[i] signals can be thought of as the OpQuad shift control signals.

LdEntry[i] signals

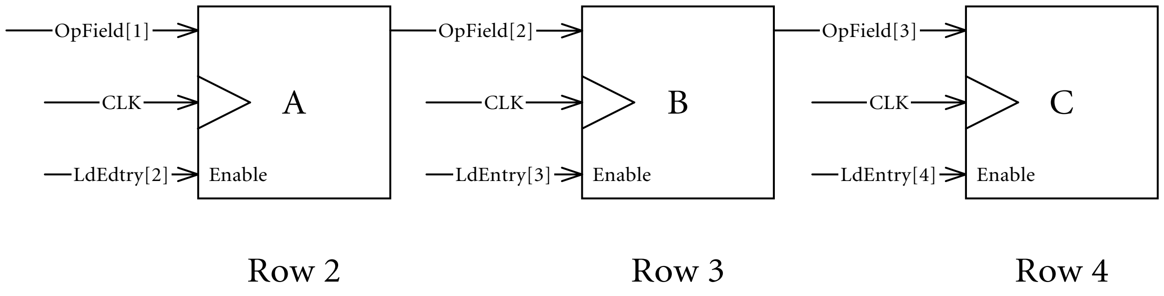


Figure 3.1 Example of One Bit of a Static Field

The rows are shifted at the rising edge of the clock signal CLK. For example, a signal LdEntry[3] either enables or disables flip-flop B, and a signal LdEntry[4] either enables or disables flip-flop C. When an OpQuad is held up in Row 4, signal LdEntry[4] is deasserted so that flip-flop C retains its value. The independence of signals LdEntry[i] allows filling of empty OpQuad entries above a held-up OpQuad. For example, even though an OpQuad is held up in Row 4, signal LdEntry[3] can be asserted so that a value OpField[2] from Row 2 shifts into Row 3 at the rising edge of clock signal CLK. Empty rows may result if the decoder is unable to provide an OpQuad every cycle—due to a branch target cache miss, for example. Empty rows are created only as a result of the decoder being unable to provide an OpQuad and by an abort cycle. Empty rows are never created by shifting lower OpQuads down while holding higher non-empty OpQuads in place since, in such cases, all the rows below the row being held will also be held from shifting.

Notation

OpQuadY, OpQY, and OpX

We will, from time to time, use the following notation to refer to *scheduler* OpQuads: OpQuadY, where Y = 0 to 5. For example, OpQuad1 identifies the OpQuad in Row1 of the scheduler, OpQuad2 the OpQuad in Row 2, etc. Additionally, we will use the following notation to refer to scheduler Op entries: OpX, where X = 0 to 23. For example, X = 0 identifies the youngest Op in the scheduler and X = 23 identifies the oldest Op in the scheduler. Thus OpQuad4, for example, contains Op16, Op17, Op18, and Op19. Moreover, the notation OpQuadY may be abbreviated OpQY in some contexts.

Pseudo-RTL Descriptions

As was mentioned in Chapter 1 and reinforced in Chapter 2, in order to assist you in understanding exactly how the microarchitecture realizes some of its functions (as well as to encourage you to simulate various chunks of logic), we give pseudo-RTL descriptions of the operation of portions of circuitry implementing specific functions. We believe the pseudo-RTL descriptions are intuitive. The first such example will be an RTL description for the operation of the circuitry implementing the shifting of data from one static field storage element to another, as shown in Figure 3.1. Such descriptions played a very important part in the overall design and simulation of the microprocessor as described in Chapter 1. First, a word about notation.

Notation

Pseudo-RTL Descriptions

A pseudo-RTL description may use signals described in other pseudo-RTL descriptions without further explanation or reference to the other descriptions since such references should be reasonably obvious from the context. Signals given in the descriptions are asserted or active high unless expressly indicated otherwise. The following notation is used in the pseudo-RTL descriptions in this text:

Signals connected via a “Þ”, “ “, and “&” are combined as a logical AND such as could be implemented by an AND gate. Signals connected via a “+” are combined as a logical OR such as could be implemented by an OR gate. Signals connected via a “^” are combined as a logical exclusive OR such as could be implemented by an XOR gate. “~” indicates the complement or inverse of a signal such as would be provided by an inverter.

Either if (a) x = b else x = c or if x = (a) ?b:c indicate a multiplexer with an output signal x equal to signal b if signal a is asserted and an output signal x equal to c otherwise. If “else x = c” is omitted, the signal x is unaffected if signal a is low (i.e., signal a forces signal x to a value if and only if a is asserted, otherwise signal x maintains its value as determined thus far, by preceding assignments to x. Another notation which represents a multiplexer is:

```
   x = switch (A) case A1: x1
                  case A2: x2
                  ...
                  case An: an
```

where output signal x has values x1 or x2 or ... xn depending on the value of a multi-bit select signal A.

The notation “xxOp.yyy” refers to an input signal to the Op decoder indicating a value from a field yyy defined for an Op of type xxOp. For example, “RegOp.Src1” refers to bits in an operation at the same position as the Src1 field of a RegOp.

Most signals described change each clock cycle. The notation “@clk:” indicates a signal is latched into a register at an edge-of-signal clock for use in a subsequent clock cycle.

Finally, it should be clear that the logic described by the pseudo RTL-descriptions can be implemented in a variety of ways.

.

Suggested Readings

Digital and Integrated Circuit Design

For readers who have little or no background in electronic circuits, we recommend reading a copy of the small but very well done introduction to the subject by Niklaus Wirth, *Digital Circuit Design for Computer Science Students, An Introductory Textbook*, Springer-Verlag, 1995. Wirth, as many of the readers know, designed a number of popular programming languages, among them Pascal and Modula. In this book he uses an easy-to-understand hardware description language called Lola.

For electronic, electrical, and computer engineers who would like an introduction to the myriad of tools and techniques used in the design, manufacture, and test of integrated circuits, we recommend Peter R. Shepherd's book, *Integrated Circuit Design, Fabrication and Test*, McGraw-Hill, 1996.

Static Field Storage Element Shifting Operation

The following pseudo-RTL description defines the operation of circuitry for the shifting of data from one static field storage element to another, as shown in Figure 3.1.

Pseudo-RTL Description

Shifting Data from One Static Field Storage Element to Another

```
@clk: if (LdEntry[i])                // OpQuad shift control
          OpField[i] = OpField[i-1]; // conditionally shift in preceding
                                     // OpField Value
```

Dynamic Field Storage Element Operation

Dynamic fields, shown in Figure 3.2, are more complicated to handle than static fields because new data from outside the buffer may be inserted into a dynamic field while the OpQuad is being shifted or re-circulated.

Two independent events are happening here, so let's expand on them to clarify this last statement:

1. the value of a dynamic field may or may not take on a new value.
2. the OpQuad may or may not shift to the next row.

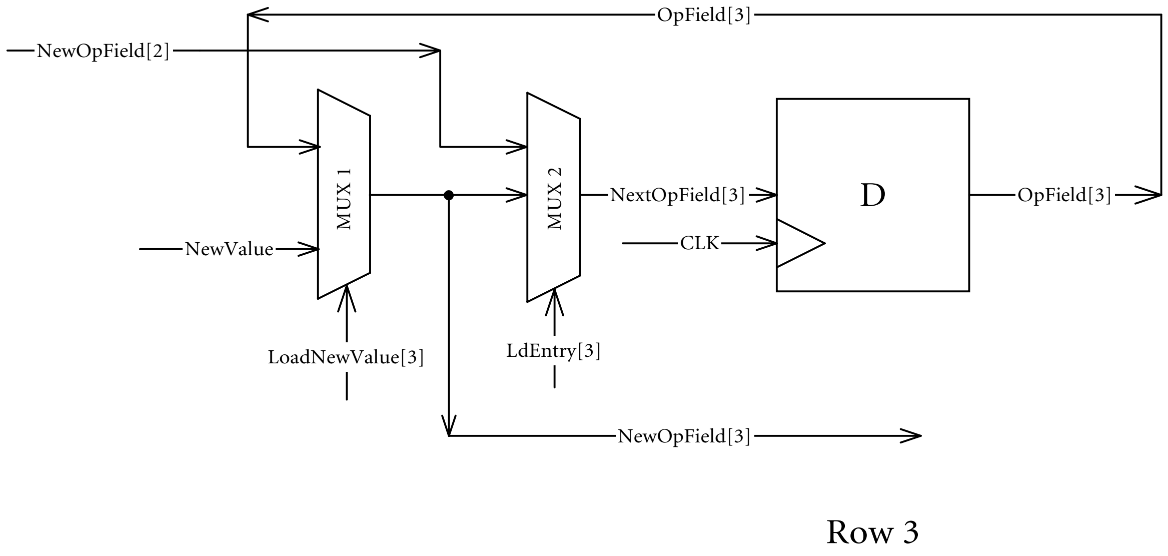


Figure 3.2 Example of One Bit of a Dynamic Field

In the example discussed here and shown in Figure 3.2, the dynamic field is called OpField and its value is stored in flip-flop D. A chunk of logic outside the scheduler generates the NewValue[23] and LoadNewValue[3] signals. If a NewValue has been generated, the LoadNewValue[3] signal selects it as the output of the MUX 1 multiplexer, otherwise the old value of OpField[3] is selected as the output. Obviously, selecting the old value is equivalent to saying that the dynamic field did not change. The output of the MUX 1 multiplexer is called NewOpField[3]. The NewOpField[3] output could have been called *Potentially_Changed_Dynamic_OpField[3]* to reflect more precisely what we just explained, but that would have been too unwieldy a name to be used in the figure.

The MUX 2 multiplexer selects whether or not the OpQuad shifts. If OpQuad[2] can shift from Row[2] to Row[3] and become OpQuad [3], then the LdEntry[3] signal selects the NewOpField[2] value from OpQuad[2] as the output of the MUX 2 multiplexer and the shift occurs at the rising edge of the clock signal CLK. Otherwise, the NewOpField[3] value is selected as the output and no shift occurs. The output of the MUX 2 multiplexer in Figure 3.2 is called NextOpField[3]. It could have been called *Potentially_Shifted_OpQuad*, to reflect more precisely what we just explained. The LdEntry[3] signal will be discussed shortly. First, it will

[23] The signal NewValue can be a signal common to all twenty-four Op entries or may be specific to each Op.

be useful to summarize the actions that occur when a valid OpQuad either shifts or does not shift:

1. *the shift occurs*: This means that OpQuad[2] shifts from Row[2] to Row[3] and OpQuad[3] shifts from Row[3] to Row[4]. By definition, all of the static and dynamic fields of OpQuad [3] take on the values of those same fields from OpQuad[2]. And, all of the static and dynamic fields of OpQuad[4] take on the values of those same fields from OpQuad[3]. Note that the value of the dynamic field OpField[3] of OpQuad[3] is sent to Row 4, via the output of the MUX 1 multiplexer.[24]
2. *the shift does not occur*: This means that OpQuad[2] stays in Row[2] only if it is valid; if it is invalid, it will be replaced with whatever is in OpQuad[1]. OpQuad[3] stays in Row[3]. This in turn means that the output of the MUX 2 multiplexer is NewOpField[3] from OpQuad[3] and not NewOpField[2] from OpQuad[2].

In summary, MUX 1 is used to conditionally choose the NextOpField value and MUX 2 is used to conditionally advance the OpQuad through the buffer. The following pseudo-RTL description defines the operation of the circuitry implementing the operations associated with dynamic fields. It should be reasonably obvious how one could have written the following description, given Figure 3.2. However, the figure represents only one implementation strategy to realize the description.

PSEUDO-RTL DESCRIPTION

Dynamic Field Operation

```
if (LoadNewValue[i])                // logic to conditionally
 NewOpField[i] = NewValue[i];       // select a value for the
else                                // dynamic field
  NewOpField[i] = OpField[i];

if (LdEntry[i])                     // logic to conditionally
  NextOpField[i] = NewOpField[i-1]; // advance the OpQuads
else
  NextOpField[i] = NewOpField[i];

@clk: OpField[i] = NextOpField[i]   // simple flip-flop
```

[24] The output of the MUX 1 multiplexer in Row[3] is an input to the MUX 2 multiplexer in Row[4].

Let’s summarize what we have learned so far. The scheduler can be viewed as a shift register in which OpQuads are loaded into the top, gradually shift downward from row to row, and eventually unloaded from the bottom. However, the scheduler is not quite exactly controlled as a true shift register since each scheduler row has its own independent shift control signal, LdEntry[i]. We will now examine these signals in more detail.

Simulators on CD-ROM

One way to test your understanding of the above concepts is to simulate the chunk of logic described by the previous two pseudo-RTL descruptions. To encourage you to do this, we have provided three simulators on the CD-ROM. Each has an extensive reference manual with it.

The LdEntry Signals: Shifting the OpQuads

The LdEntry[i] signals determine if the OpQuad in Row[i-1] shifts into Row[i]. In general, OpQuads always shift down whenever there is space below. That is, as long as there is at least one empty row below a given OpQuad, or the bottom row of the scheduler is being unloaded, then the OpQuad can shift down to the next row, the OpQuad above can shift down into the given OpQuad’s row from above, and so on. However, situations can exist that prevent OpQuads from shifting. The situations that can hold up OpQuads basically fall into five categories:

1. OpQuads containing Ops that are dependent on status flags as part of their processing, i.e., BRCOND Ops and cc-dependent RegOps.
2. OpQuads that contain nonabortable RegOps.
3. OpQuads that are being considered for commitment by the OCU and contain one or more Ops that have not yet completed execution.
4. OpQuads containing multiple StOps in an OpQuad.[25]
5. OpQuads above the bottom row of the scheduler that cannot shift.

We will examine each of these categories separately:

1. *1st Category*: It was pointed out in the section titled “The Scheduler” beginning on page 128 that “… some Ops (such as the evaluation of conditional branches and RegOps that depend on status

[25] More than one StOp in an OpQuad does not guarantee a hold up since, as we will see later, the OCU can look ahead into the second OpQuad row from the bottom to get a “head start” on committing StOps.

flags) are executed when the Ops reach a particular row of the buffer." Conditional branches, for example, are held in Row[3] until they are resolved and only then are they allowed to advance. If the branch is resolved as correctly predicted, processing proceeds normally. However, if the branch is resolved as mispredicted in Row[3], an appropriate restart signal is asserted that causes the upper portion of the machine to be flushed and then restarted to begin fetching instructions from the not-predicted path. The flushing of the upper portion is done as part of the resolution of the branch versus as part of the abort cycle since the abort cycle will be initiated later when the OpQuad reaches the bottom of the scheduler. If Ops are generated before the lower portion of the machine can accept them, then these Ops are held until they can be loaded into the scheduler. The BrAbort signal is asserted when the OpQuad containing a mispredicted branch reaches the bottom row of the scheduler. CC-dependent RegOps are also held in Row[3] until the necessary status flag operand value(s) are successfully obtained, and only then are they allowed to advance or shift down into lower rows.

2. *2nd Category*: There are four nonabortable RegOps, WRDR, WRDL, WRxxx, and WRDLP. Nonabortable RegOps that cannot yet proceed into the Execution Stage of the RegOp pipeline are held when they reach Row[4] until they can execute.
3. *3rd Category*: Because of the out-of-order execution nature of the processor, or due simply to execution latencies and serial dependencies between Ops, an Op might reach the bottom row of the scheduler before the other Ops associated with the OpQuad have completed execution. It must wait there while the other Ops are still executing.
4. *4th Category*: Consider a situation in which there are four StOps, two in each of two consecutive OpQuads. The first StOp will commit from OpQuad4, the second from OpQuad5 (the first OpQuad retires without delay), the third and fourth from OpQuad5 and thus shifting of the second OpQuad out of the scheduler is delayed for one cycle.
5. *5th Category:* There can be instances when OpQuads cannot shift because all rows below contain valid OpQuads and they are not able to shift. Conversely, an OpQuad is generally able to shift if there is an invalid OpQuad immediately below it or if all of the OpQuads below it are able to shift down.

The pseudo-RTL description that follows summarizes the equations determining which OpQuad can advance at the next clock cycle boundary. The code reflects the five categories discussed above as constraints on the

LdEntry signals that advance the OpQuads. While it may be apparent to some readers, what might not be obvious to others that there is the potential for some "deadlock" situations to occur. For example, you do not want to hold OpQuad[4] or OpQuad[5] if a branch has been detected as mispredicted or it will never reach the bottom row of the scheduler to initiate the abort cycle.

A few words about the notation for signals in the description are in order. The LdEntry[i], SC_MisPred, BrAbort, and trap pending flag have all been introduced in the preceding paragraph. The Q4PendLdStAbort signal indicates that there is a LdOp or a StOp in OpQuad4 which has a pending exception waiting to be recognized by the OCU (once it gets into OpQuad[5]) and then cause an abort cycle. The OpQV[i], i = 0 to 5, signals represent the state of valid bit of OpQuad[i]. The OpQRetire signal originates from the OCU and, if asserted, indicates when a valid OpQuad in the bottom scheduler can be retired. The HoldOpQ45 signal, if asserted, holds up both OpQuad[4] and OpQuad[5]. HoldOpQ3 and HoldOpQ4A correspond to BRCOND resolution and cc-dependent RegOps, both of these situations are in Category 1 above. The HoldOpQ4B signals correspond to Category 2 above. Both Category 3 and Category 4 are handled via the OpRetire signal. Category 5 is handled via the OpV[i] signals.

The scheduler generates signals that indicate whether it will be able to accept a new OpQuad at the end of the current cycle. This is shown in the following pseudo-RTL description.

Pseudo-RTL Description

Advancing OpQuads in the Scheduler

```
HoldOpQ45 = (HoldOpQ3 | HoldOpQ4A | HoldOpQ4B) &
            ~(SC_MisPred | Q4PendLdStAbort | "trap pending")

LdEntry[5] = (OpQRetire | ~OpQV[5]) & ~HoldOpQ45

LdEntry[4] = (OpQRetire | ~OpQV[5] | ~OpQV[4]) & ~HoldOpQ45

LdEntry[3] = LdEntry[4] | ~OpQV[3]

LdEntry[2] = LdEntry[4] | ~OpQV[3] | ~OpQV[2]

LdEntry[1] = LdEntry[4] | ~OpQV[3] | ~OpQV[2] | ~OpQV[1]

LdEntry[0] = LdEntry[4] | ~OpQV[3] | ~OpQV[2] | ~OpQV[1] |
             ~OpQV[0] | BrAbort
```

The signal "~LdEntry[0]" indicates that OpQuad[0] is full and not shifting and thus the scheduler does not have enough room to accept a new OpQuad. The signal "SC_MisPred & ~BrAbort" indicates that there is a pending mispredicted branch and while the upper portion of the processor has been restarted and may have a new OpQuad ready, that the scheduler cannot accept it until the abort cycle is started.

PSEUDO-RTL DESCRIPTION

SchedFull and SchedEmpty Signals

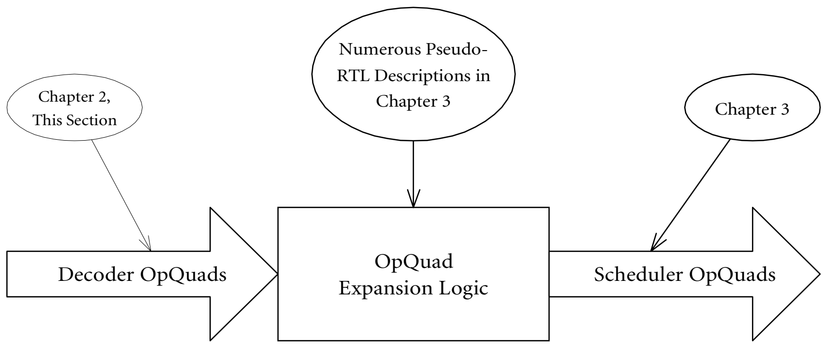
```
SchedFull = ~LdEntry[0] | SC_MisPred & ~BrAbort

SchedEmpty = ~(OpQV[0] | OpQV[1] | OpQV[2] | OpQV[3] | OpQV[4] |OpQV[5])
```

We will now look at the static, dynamic, and OpQuad fields in more detail.

STATIC AND DYNAMIC FIELDS

The static and dynamic fields in an entry are shown in Table 3.1 and Table 3.2 respectively. These fields are related to but not identical to the fields of the associated *decoder* OpQuad Op formats shown in Table 2.22, "Decoder OpQuad LdOp and StOp Format," Table 2.30, "Decoder OpQuad RegOp Format," Table 2.38, "Decoder OpQuad SpecOp Format," and Table 2.42, "Decoder OpQuad LIMM Op Format." Indeed, if we return to the following diagram given in Chapter 2, we see that the relationship between these fields is established by the OpQuad expansion logic:



The eleven static fields require a total of 45 bits per Op entry. The sixteen dynamic fields require a total of 65 bits per Op entry.

Table 3.1 Static Fields Per Op Entry

Field Identifier	Bits/Entry
Type[2:0]	3
Imm	1
Src1Reg[4:0]	5
Src2Reg[4:0]	5
DestReg[4:0]	5
SrcStReg[4:0]	5
Src1BM[1:0]	2
Src2BM[1:0]	2
Src12BM[2]	1
SrcStBM[2:0]	3
OpInfo[12:0]	13

Table 3.2 Dynamic Fields per Op Entry

Field Identifier	Bits/Entry
State[3:0]	4
ExecX	1
DestBM[2:0]	3
DestVal[31:0]	32
StatMod[3:0]	4
StatVal[7:0]	8
OprndMatch LUsrc1	1
OprndMatch LUsrc2	1
OprndMatch SUsrc1	1
OprndMatch SUsrc2	1
OprndMatch SUsrcSt	1
OprndMatch RUXsrc1	1
OprndMatch RUXsrc2	1
OprndMatch RUYsrc1	1
OprndMatch RUYsrc2	1
DBN[3:0]	4

Additionally, the OpQuad fields, which are also stored in the buffer on a per OpQuad basis, are shown in Table 3.3. The thirteen OpQuad fields require a total of 138 bits.

Table 3.3 OpQuad Fields per OpQuad

Field Identifier	Static/Dynamic	Bits/OpQuad
Emcode	static	1
Eret	static	1
FaultPC[31:0]	static	32
BPTInfo[14:0]	static	15
RASPtr[2:0]	static	3
LimViol	dynamic	1
OpQV	dynamic	1
OpQFpOp	static	1
ILen0[2:0]	static	3
Smc1stAddr	static	20
Smc1stPg	static	20
SMC2ndAddr	static	20
Smc2ndPg	static	20

Thus, a scheduler OpQuad requires a total of 45*4 + 65*4 + 180 = 578 bits.

The initial values of the static and dynamic fields depend on the corresponding Op loaded into that entry. As mentioned earlier, the OpQuad Expansion Logic modifies some fields from the Op based on other fields, derives new fields from existing ones, replaces some fields with physically different fields, and passes a few fields through unchanged. The OpQuad fields are generated from information corresponding to the OpQuad as a whole.

An Op Entry's Static Fields in More Detail

Each scheduler entry contains the following eleven static fields:

1. Type[2:0]
2. Imm
3. Src1Reg[4:0]
4. Src2Reg[4:0]
5. SrcStReg[4:0]
6. DestReg[4:0]

7. Src1BM[1:0]
8. Src2BM[1:0]
9. Src12BM[2]
10. SrcStBM[2:0]
11. OpInfo[12:0]

In the following discussion, all signals are actively high. Before proceeding, we make an additional comment about notation.

Notation

Decoder OpQuad Field Notation

There are many references to various decoder OpQuad fields in the pseudo-RTL description. We believe the notation identifying them, such as:

1. RegOp.Src1 (the Src1 field of the RegOp)
2. LdStOp.Data (the Op Data field of the LdOp or StOp)
3. SpecOp.Dest (the Op Dest field of the SpecOp)
4. LdOp.Type[1] (bit 1 of the Type field of the LdOp)

is quite intuitive if the reader examines the decoder OpQuad Op formats cited above.

Static Field Type(2:0)

The static field Type[2:0] specifies the type of Op for the entry, particularly for issue selection purposes. Possible types include: a SpecOp; a LdOp; a StOp which references memory or generates a faultable address; a RegOp executable only by RUX; and a RegOp executable by either RUX or RUY. Floating-point operations (FpOps) are a type of SpecOp executed by the floating-point unit. This can be summarized in the following table:

Table 3.4 Op Type Specified by the Type Field

Type(2:0)	Type of Op
000	a SpecOp—not issued to an execution unit
010	a LdOp—issued to the Load Unit
10x	applies to all StOps
100	a StOp that does not reference memory—issued to the Store Unit
101	a StOp that references memory or at least can result in a memory fault—issued to the Store Unit
110	a RegOp that can only be executed by RUX—issued to RUX
111	a RegOp that can be executed by RUX or RUY—issued to either RUX or to RUY

The pseudo-RTL description that follows defines the chunk of circuitry in the OpQuad Expansion Logic that generates a value for the static field Type. In the equations in the description, fields in the *scheduler* OpQuad appear on the left-hand side of the equations and fields from the *decoder* OpQuad appear on the right-hand side. For example, in the description below the equation Type[2] = LdStOp.Type[3] means that bit three of the Type field of the *decoder* LdStOp is assigned to bit two of the static field Type of the *scheduler* Op entry. "RUYD" is a signal from a special register bit that inhibits use of the second register unit RUY for silicon debugging purposes; see Table 2.11 on page 88.

PSEUDO-RTL DESCRIPTION

Static Field Type

```
switch(OpId)
  case RegOp:
    Type[2:1] = 2'b11
    Type[0] = ~(RegOp.R1 | RUYD)
  case LdStOp:
    Type[2] = LdStOp.Type[3]
    Type[1] = ~LdStOp.Type[3]
    Type[0] = LdStOp.Type[3] & ~(LdStOp.Type[2] &
              LdStOp.Type[1])
  default:
    Type[2:0] = 3'b000
```

Static Field Imm

For RegOps, the static field Imm indicates that the Src2 operand is an immediate value (being temporarily held in the DestVal field of the Op entry) instead of a register. For LdStOps, the static field Imm is not used.

PSEUDO-RTL DESCRIPTION

Static Field Imm

```
Imm = RegOp.I  // don't care if not RegOp
```

Static Fields Src1Reg(4:0), Src2Reg(4:0), & SrcStReg(4:0)

Some Ops can have up to two input values (obtained from registers). StOps, which actually write to memory, have a third input value, the data to be stored.

Articles on CD-ROM

Chapter 3 of the AMD application note, *AMD-K6 3D Processor Code Optimization*, gives several examples of address register operands, data register operands, and store data register operands. This application note is on the CD-ROM.

Fields Src1Reg[4:0], Src2Reg[4:0], and SrcStReg[4:0] hold register numbers identifying the registers which an Op uses. Src1Reg[4:0] holds the register number of the first source operand Src1, Src2Reg[4:0] the register number of the second source operand Src2, and SrcStReg[4:0] the register number of the store data operand in the case of StOps. The following three pseudo-RTL descriptions define the circuitry in the OpQuad Expansion Logic that generates values for the static fields Src1Reg, Src2Reg, and SrcStReg:

PSEUDO-RTL DESCRIPTIONS

Static Fields Src1Reg, Src2Reg, and SrcStReg

```
// Field Src1Reg
if (OpId = RegOp)
  Src1Reg[4:0] = RegOp.Src1
  Src1Reg[2] &= ~(LdStOp.DSz=1B)    // do byte register conversion
else
  Src1Reg[4:0] = {1'b0,LdStOp.Base} // don't care if not RegOp or LdStOp

// Field Src2Reg
if (OpId = RegOp)
  Src2Reg[4:0] = RegOp.Src2
  Src2Reg[2] &= ~(LdStOp.DSz=1B)    // do byte register conversion
else
  Src2Reg = {1'b0,LdStOp.Index}     // don't care if not RegOp or LdStOp

// Field SrcStReg
SrcStReg[4:0] = LdStOp.Data
SrcStReg[2] &= ~(LdStOp.DSz=1B & LdStOp.DataReg=t0)
// don't care if not StOp
```

Static Field DestReg(4:0)

Static Field DestReg[4:0] holds a register number identifying the destination register of the Op. The following pseudo-RTL description defines the circuitry in the OpQuad Expansion Logic that generates a value for the static field DestReg:

Pseudo-RTL Description

Static Field DestReg[4:0]

```
if (OpId = LIMMOp)
  DestReg[4:0] = {1'b0,LIMMOp.Dest}
elseif ((OpId = LdStOp) & (LdStOp.Type = STUPD))
  DestReg[4:0] = {1'b0,LdStOp.Base}
else
  DestReg[4:0] = LdStOp.Data
  DestReg[2] &= ~(LdStOp.DSz=1B) // do byte register conversion
                                 // don't care if non-STUPD StOp
```

Static Fields Src1BM(1:0), Src2BM(1:0), & Src12BM(2)

The x86 instruction can operate on individual bytes and 16-bit words as well as 32-bit double words and, correspondingly, can modify just parts of 32-bit registers. The K6 3D's microarchitecture reflects this ability. Static fields Src1BM[1:0], Src2BM[1:0], and Src12BM[2] specify the sizes and locations of the operands. The "BM" is used as an abbreviation for the phrase "byte marks" and these three fields indicate which bytes of operand registers Src1 and Src2 must be "valid" for execution of the Op that will use values from these registers—i.e., which fields in the source registers must have correct, up-to-date values in them so the Op can proceed.

Src12BM functions as both Src1BM[2] and Src2BM[2]. A "0" in Src12BM means that the high-order 16-bits of neither source register will be used. A "1" in Src12BM[2] specifies that the high-order 16-bits of *both* source registers will be used. That is, a "1" for a BM indicates that the corresponding register part *will* be used as it is presumed to be "valid." The Src!BM and Src2BM fields indicate if the byte positioned at bit [15:8] in the Src register or the byte positioned at bits [7:0] will be used if they contain a valid value. The byte positioned at [15:8] is specified by a "1" and the byte positioned at [7:0] is specified by a "0." Thus, bits 2, 1, and 0 of the SrcBM fields correspond to bits [31:16], [15:8], and [7:0] respectively. The following pseudo-RTL description defines the OpQuad Expansion Logic circuitry that generates values for the Src1BM[1:0], Src2BM[1:0], and Src12BM[2] fields.

Pseudo-RTL Descriptions

Static Fields Src1BM, Src2BM, and Src12BM

```
if (OpId = RegOp)
  Src1BM[0] = ~(RegOp.DSz = 1B) | ~RegOp.Src1[2]
  Src1BM[1] = ~(RegOp.DSz = 1B) |  RegOp.Src1[2]
  Src2BM[0] = ~(RegOp.DSz = 1B) | ~RegOp.Src2[2] |  RegOp.I
  Src2BM[1] = ~(RegOp.DSz = 1B) |  RegOp.Src2[2] & ~RegOp.I
  if (RegOp.Type = 6'b10001x)
    Src2BM[1] = Src1BM[1] = 1'b0 // if ZEXT,SEXT
    Src12BM[2] = (RegOp.DSz = 4B)
    if ((RegOp.Type = 6'b10001x) | (RegOp.Type = 6'b111x00))
      Src12BM[2] = 1'b0  // if ZEXT,SEXT,CHKS

else// else LdStOp or don't care
  Src1BM[1:0] = Src2BM[1:0] = 2'b11
  Src12BM[2] = (LdStOp.ASz = 4B)  // don't-care if LIMM

if (LdStOp.Type = 4'bx0xx) { // STxxx Ops
  SrcStBM[0] = ~(LdStOp.DSz=1B) | ~LdStOp.Data[2]
  SrcStBM[1] = ~(LdStOp.DSz=1B) |  LdStOp.Data[2]
  SrcStBM[2] =  (LdStOp.DSz=4B)
} else
  SrcStBM[2..0] = 3'b000// CDA,CIA,LEA Ops
// don't care if not StOp
```

Static Field SrcStBM(2:0)

Static Field SrcStBM[2:0] indicates which bytes of the store data operand are required for completion of a StOp. The bit correspondence is the same as for Src1BM or Src2BM. The following pseudo-RTL description defines the circuitry in the OpQuad Expansion Logic that generates a value for the static field SrcStBM:

Pseudo-RTL Description

Static Field SrcStBM

```
if (LdStOp.Type = 4'bx0xx) // STxxx Ops
  SrcStBM[0] = ~(LdStOp.DSz = 1B) | ~LdStOp.Data[2]
  SrcStBM[1] = ~(LdStOp.DSz = 1B) |  LdStOp.Data[2]
  SrcStBM[2] =  (LdStOp.DSz = 4B)
else
  SrcStBM[2:0] = 3'b000// CDA,CIA,LEA Ops
  // don't care if not StOp
```

Static Field OpInfo(12:0)

The static field OpInfo[12:0] holds additional information about the Op for either the execution units or the OCU depending on whether the operation is executable or not. OpInfo is the union of three possible field definitions, depending on whether the Op is a RegOp, a LdStOp, or a SpecOp:

1. for a RegOp, field OpInfo contains a concatenation of the following bits from the Op template: six bits from the Op Type field; four bits from the Op Ext field; the Op R1 field; and two bits indicating an effective data size DataSz for the operation.

Table 3.5 OpInfo Data for a RegOp

Op Template	Description
Type[5:0]	copy of the original Op Type field
Ext[3:0]	copy of the original Op Ext field
R1	copy of the original Op R1 field
DataSz[1:0]	effective data size of the Op (one, two, or four bytes)

2. for a LdStOp, field OpInfo contains a concatenation of the following bits from the Op template: four bits from the Op Type field; two bits from the Op ISF field; four bits from the Op Seg field; two bits indicating the effective data size DataSz for the operation; and a bit AddrSz indicating a 16-bit or a 32-bit effective address size for the address calculation.

Table 3.6 OpInfo Data for a LdStOp

Op template	Description
Type[3:0]	copy of the original Op Type field
ISF[1:0]	copy of the original Op ISF field
Seg[3:0]	copy of the original Op Seg field
DataSz[1:0]	effective data size for the memory transfer
AddrSz	effective address size for the address calculation

3. for a SpecOp, the OpInfo field contains a concatenation of the following bits from the Op template: four bits from the Op Type field and five bits from the Op CC field.

Table 3.7 OpInfo Data for a SpecOp

Op template	Description
Type[3:0]	copy of the original Op Type field
CC[4:0]	copy of the original Op CC field

The following pseudo-RTL description defines the OpQuad Expansion Logic circuitry that generates a value for the OpInfo Static field:

Pseudo-RTL Description

Static Field OpInfo

```
OpInfo[12] = Op[35]
              // prevent LIMM from looking like various exception Ops

OpInfo[11:8] = (OpId = LIMMOp) ? 4'b1111 : Op[34:31]
OpInfo[7:0] = {Op[30:25],Op[23:22]}
```

An Op Entry's Dynamic Fields in More Detail

The entry's dynamic fields are initialized by the operation decoder but can then change during execution of Ops. Typically, each entry contains logic for changing the values in dynamic fields as required. Each scheduler entry contains the following eight dynamic fields:

1. State[3:0].
2. Exec1.
3. DestBM[2:0].
4. DestVal[31:0].
5. StatMod[3:0].
6. StatVal[7:0].
7. OprndMatch_XXsrcY.
8. DBN[3:0].

These fields are discussed in the following paragraphs:

Dynamic Field State(3:0)

The dynamic field State[3:0] indicates an operation's execution state with respect to the execution unit pipelines. In Figure 2.16 on page 165 and Figure 2.19 on page 168, the stages S2, S1, and S0 are alternate signal names for State[3:0]. The State[3:0] bits are updated as the Op is successfully issued or advances out of a pipeline stage. The updating can be viewed as shifting a field of ones across four bits.

Table 3.8 INTERPRETATIONS OF THE STATE FIELD

S3	S2	S1	S0	Indicates the Op is
0	0	0	0	unissued / not yet issued
0	0	0	1	in operand fetch stage
0	0	1	1	in execution stage 1
0	1	1	1	in execution stage 2
1	1	1	1	completed

Most Ops enter the scheduler with field State set to 0000 (unissued). The dynamic field State changes after the operation issues to an execution pipeline. Upon completion of the execution pipeline, State is set to 1111 (completed) while the Op awaits to be committed or retired. The state field of every scheduler entry is set to 1111 during abort cycles. Some Ops (e.g., load constant LDK or load 32-bit immediate LIMM) have an initial state field value of 1111 and thus are already completed when loaded into the scheduler. The following pseudo-RTL description defines the circuitry in the OpQuad Expansion Logic that initializes dynamic field State and the circuitry in the scheduler entries that modify field State during execution of the associated operation. Note that "OPQV" in the pseudo-RTL descriptions for the dynamic fields is the OpQuad Valid bit associated with the incoming decoder OpQuad and not the OpQV field of a scheduler OpQuad. The OpQuad Expansion Logic initializes field State[3:0] either as 0000 (unissued) or 1111 (completed) according to the OpId field. Signal SC_Abort is asserted to abort execution of Ops currently in the scheduler. The signal "Issue Op[i] to XXX" are generated during the Op issue selection scan process; see the section titled "Issue Selection Logic" beginning on page 228.

PSEUDO-RTL DESCRIPTION

Dynamic Field State

```
State[3:0] = (~OpQV |
              (OpId = SpecOp) &
              ((SpecOp.Type = LDKxx) | (SpecOp.Type = LDxHA)) |
              (OpId = LIMMOp)) ? 4'b1111 : 4'b0000

Field State (signals S0, S1, S2, and S3) changes during operation execution
as follows.

if (S0Enbl) S0 = ~BumpEntry | SC_Abort
if (S1Enbl) S1 = S0 & ~BumpEntry | SC_Abort
if (S2Enbl) S2 = S1 | SC_Abort
if (S3Enbl) S3 = S2 | S1 & RU | SC_Abort

BumpEntry = RU & ~S1 & S0 & (Exec1 & BumpRUX | ~Exec1 & BumpRUY)

S0Enbl = “Issue Op[i] to “ & CHP_LUAdv0 |
         “Op[i] to SU “ & CHP_SUAdv0  |
         “Issue Op[i] to RUX” & CHP_RUXAdv0 |
         “Issue Op[i] to RUY” & CHP_RUYAdv0 |
         SC_Abort | BumpEntry

S1Enbl = LU & CHP_LUAdv0 |
         SU & CHP_SUAdv0 |
         RU & (Exec1 & CHP_RUXAdv0 | ~Exec1 & CHP_RUYAdv0) |
         SC_Abort

S2Enbl = LU & CHP_LUAdv1 | SU & CHP_SUAdv1 | RU | SC_Abort
S3Enbl = LU & CHP_LUAdv2 | SU & CHP_SUAdv2 | RU | SC_Abort
```

Abort Handling (revisited)

The signal SC_Abort in the above pseudo-RTL description warrants special attention. When an abort cycle occurs, the entire scheduler is flushed. All OpQuad entries are invalidated by clearing all of the OpQuad Valid fields (OpQV) and certain fields of all Op entries are also cleared to innocuous values. The latter is necessary since OpQV only affects the control of scheduler OpQuad entry loading and shifting. All other operations within the scheduler ignore OpQV and simply assume that Op entries are always valid and sufficiently well defined.

DESIGN NOTE

Representation of an Invalid Op

An invalid Op within the scheduler is represented as a valid but innocuous Op. Its State field is set to completed so the Op will not be executed. Its DestBM and StatMod fields are set to indicate that it does not modify any register bytes or status flags. All other fields, in these circumstances, can have any values without causing any "harm" (side effects). An invalid Op is effectively a NoOp.

One important aspect of abort cycle handling within the scheduler occurs after mispredicted BRCOND Ops. In this case, a new OpQuad may be loaded into the scheduler during the abort cycle. This OpQuad is not associated with any of the outstanding OpQuads that all need to be flushed. It is logically the first new OpQuad after the abort. In all other cases, there will be a delay to the reception of the first new OpQuad after abort cycles due to exception conditions.

As discussed in the section titled "Loading the Scheduler" beginning on page 184, in the section titled "Static Field Storage Element Shifting Operation" beginning on page 190 and the section titled "Dynamic Field Storage Element Operation" beginning on page 190, the storage elements within the scheduler are fully synchronous and do not change state in response to inputs until the next cycle boundary. Thus, the following sequence of events occur at the end of the abort cycle. First, certain Op entry fields are changed to innocuous values. Then all, some, or none of the OpQuad entries shift down one position and a new OpQuad is loaded into the top scheduler entry. The shifting of any OpQuads other than OpQuad[0] during an abort cycle is a "don't care" situation. In the case of exception-related aborts, this new OpQuad is also invalidated. In the case of BRCOND-related aborts, this new OpQuad is allowed to be valid and reloading of the top scheduler OpQuad entry is forced.

To improve the clock frequency, there is both an "early" and a "late" version of the abort signal. The late version is logically the same as the early version, but delayed by one cycle using a flip-flop. The late version is called SC_Abort, and the early version is called SC_EAbort. SC_EAbort is used to flush the scheduler immediately; SC_Abort is used to flush the execution pipelines—this is not performance critical since there will be at least two cycles after SC_EAbort before an Op can possibly be ready to enter Stage 1 of a pipeline. The SC_Abort signal was "split off" of SC_EAbort to reduce the fanout/loading on a critical signal and to also make the longer distance usages non-timing critical. In short, this was done for timing improvement on a critical signal.

Dynamic Field Exec1

If the Op is a RegOp, the Exec1 field indicates that register unit RUX (versus RUY) is executing it. This field is set when the Op has successfully been issued to RUX or RUY. The OpQuad Expansion Logic initializes the Exec1 field to low (it is actually a “don’t care” before the RegOp is issued). The following pseudo-RTL description defines the logic which sets and changes field Exec1. The signals “Issue Op[i] to RUX” are generated during the Op issue selection scan process; see the section titled “Issue Selection Logic” beginning on page 228.

PSEUDO-RTL DESCRIPTION

Dynamic Field Exec1

```
if (S0Enbl) Exec1 = “Issue Op[i] to RUX”
```

Dynamic Field DestBM(2:0)

The dynamic field DestBM[2:0] specifies which bytes of the register specified by the DestReg field are modified by the Op. DestBM[2], DestBM[1], and DestBM[0] correspond to bits [31:16], [15:8], and [7:0], respectively. The DestBM field is initialized by the operation decoder and may be cleared during an abort cycle. The logic associated with dynamic field DestBM is given in the following pseudo-RTL description. As in the case of the Src1/2BM fields, “BM” is used as an abbreviation for “byte marks.” The equations shown are for integer results. There are similar 64-bit MDestVal fields (one per pair of Ops) for MMX/3D register results.

Dynamic Field DestVal(31:0)

The dynamic field DestVal[31:0] holds the register result value which has resulted from execution of the Op and is to be committed to DestReg. DestBM indicates which bytes of the result value are valid after the execution of the Op. The DestVal field is loaded when the Op completes execution stage 1 or 2 (depending on the type of Op). For non-executed Ops (e.g., the load constant operation LDK), DestVal is initialized with the appropriate register result value. DestVal can be used for temporary storage before register results are stored when an Op is completed. DestVal initially holds immediate values for RegOps, displacement values for LdStOps, and the alternate (sequential or target) branch program counter (PC) value for a BRCOND Op.

Pseudo-RTL Description

Dynamic Field DestBM

```
Initialization by OpQuad Expansion Logic:

if (OpId=LIMMOp)
  if (LIMMOp.DestReg = t0)
    DestBM[2:0] = 3'b000
  else
    DestBM[2:0] = 3'b111
elseif (OpId=LdStOp LdStOp.Type = STUPD)
  DestBM[1:0] = 2'b11
  DestBM[2] = (LdStOp.ASz=4B)
else
  DestBM[0] = ~(LdStOp.DSz=1B) | ~LdStOp.Data[2]
  DestBM[1] = ~(LdStOp.DSz=1B) | LdStOp.Data[2]
  DestBM[2] = (LdStOp.DSz=4B)

if (~OpQV | (DestReg[4:0] = 5'b01111) | // invalid or dest is t0
    ((OpId = LdStOp) & (LdStOp.Type = ST/STF))  // stores have no dest reg
  DestBM = 3'b000

if (SC_Abort)
  DestBM = 3'b000
```

The DestVal field plays an important role in the K6's implicit renaming strategy. The OpQuad Expansion Logic circuitry used to initialize the DestVal field and the scheduler circuitry logic associated with dynamic field DestVal is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

Dynamic Field DestVal

```
The OpQuad Expansion Logic generates the DestVal field according to
the following logic:

DestVal[31:0] = switch(OpId)
                  case RegOp:  sext(RegOp.Imm8)
                  case LdStOp: sext(LdStOp.Disp8)
                  case LIMMOp: {LIMMOp.ImmHi[16:0],LIMMOp.ImmLo[16:0]}
                  case SpecOp: if (SpecOp.Type=BRCOND & ~DEC_OpQSel_E)
                                 DEC_AltNextIPC[31:0]
                               else
                                 sext(SpecOp.Imm17)

Following execution of the Op, the DestVal field changes as follows:

if ((~S2 | LU) & ~S3 & S1)
  DestVal[31:0] = switch (Type)
                    case LU:            DC_DestRes
                    case SU:            SU1_DestRes
                    case (RU & Exec1):  RUX_DestRes
                    case (RU & ~Exec1): RUY_DestRes

where signals DC_DestRes, SU1_DestRes, RUX_DestRes, and RUY_DestRes are the
LU, SU, RUX, and RUY result buses, respectively.
```

Dynamic Field StatMod(3:0)

Status flag bits EZF, ECF, OF, SF, AF, PF, and CF may be modified by RegOps. This field specifies which groups of status flags can be modified by the Op as shown in the following table:

Table 3.9 STATUS FLAGS GROUPS SPECIFIED BY THE STATMOD FIELD

StatMod bit	Status Flags Groups that can be Modified by RegOps
3	{EZF, ECF}
2	{OF}
1	{SF, ZF, AF, PF}
0	{CF}

EZF and ECF are separate “zero” and “carry” flags for use within OpQuad Sequences. When set, they are set in the same way as the architectural ZF and CF flags. The StatMod field is initialized to all zeroes for non-RegOp Ops and is cleared during abort cycles. The logic associated with dynamic field StatMod is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

Dynamic Field StatMod

```
Initialization by OpQuad Expansion Logic:

StatMod[3:0] = (~OpQV & (OpId=RegOp) & RegOp.SS) ? RegOp.Ext : 4'b0000

Logic in the scheduler clears field StatMod during an abort:

if (Exec1 & ~S3 & S1 & RUX_NoStatMod | SC_Abort)
  StatMod[3:0] = 4'b0000
```

Dynamic Field StatVal(7:0)

Like the StatMod field, the StatVal field is significant only for RegOps. The StatVal dynamic field stores the Op’s status flag results value which are to be committed to status register EFLAGS. StatMod indicates which bits are valid after the RegOp completes execution stage 1. The StatVal[7:0] field is loaded when the RegOp completes execution stage 1. The logic associated with dynamic field StatVal is given in the following pseudo-RTL description. Note that there are no two-cycle RegOps that produce status flag results.

PSEUDO-RTL DESCRIPTION

Dynamic Field StatVal

```
Field StatVal is initially set to zero (i.e StatVal = 8'b00000000) and
changes when a RegOp completes execution pipeline Stage 1.

if (~S3 & S1)
   StatVal[7:0] = Exec1 ? RUX_StatRes[7:0] : RUY_StatRes[7:0]
```

Dynamic Fields OprndMatch_XXsrcY

This set of dynamic fields is associated with the control of transient information that is passed between two adjacent pipeline stages; see, for example, Figure 2.13 on page 160. In the notation, OprndMatch_XXsrcY, XX is either the Load Unit, the Store Unit, RUX, or RUY, and Y is either 1 or 2.

The logic associated with dynamic field OprndMatch_XXsrcY is given in the following pseudo-RTL description:

Pseudo-RTL Description

Dynamic Fields OprndMatch_XXsrcY

```
match with operand XXsrcY:
OprndMatch_XXsrcY =(busReg[4:0] = DestReg[4:0]) &
                   (busBM[1] & DestBM[1] | busBM[0] & DestBM[1])

where XXsrcY takes on the values LUsrc1, LUsrc2, SUsrc1, SUsrc2, RUXsrc1,
RUXsrc2, RUYsrc1, and RUYsrc2, and “bus” refers to OprndInfo_XXsrcY. The
byte mark checking does not include BM[2], as a simplification, since (BM[2]
= 1'b1) implies (BM[1] & BM[0]); thus, if (bus.BM[2] = 1'b1), then a match
will be signaled irrespective of DestBM[2].
```

Dynamic Field DBN(3:0)

The DBN[3:0] dynamic field holds four data breakpoint status bits Bn (for n = 0 to 3) for a LdStOp. This field is initially all zeroes. When the associated LdStOp executes, the breakpoint bits from the appropriate execution unit are recorded for later trapping. Field DBN is initialized to zero (DBN[3:0] = b0000). Scheduler circuitry changes it during execution as shown in the following pseudo-RTL description:

Pseudo-RTL Description

Dynamic Field DBN

```
if ((AdvLU2 | AdvSU2) & ~S3 & S2)
  DBN[3:0] = (DBN_LU[3:0] & LU) | (DBN_SU[3:0] & SU)
```

The OpQuad Fields in More Detail

In addition to the static and dynamic fields for each Op entry in a row, the scheduler contains fields that are associated with the OpQuad as a whole. Most of these OpQuad fields are static; however, some are dynamic. Logic in each row of the scheduler changes the dynamic OpQuad fields as required. We will now examine the OpQuad fields that were given in Table 3.3 on page 198 in more detail.

OpQuad Field Emcode

The Emcode field indicates if the OpQuad was fetched from the OpQuad ROM or if it was generated from the hardware decoders. The scheduler logic associated with field Emcode field, which is a static field, is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

OpQuad Field Emcode

```
Emcode = DEC_OpQSel_E | DEC_Vec2Emc
  // treat initial vectoring OpQuad as part of an
  // OpQuad Sequence
```

OpQuad Field Eret

The Eret field indicates the OpQuad was fetched from the OpQuad ROM and that it is marked as the last OpQuad in an OpQuad Sequence. The scheduler logic associated with the Eret field, which is a static field, is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

OpQuad Field Eret

```
Eret = DEC_OpQSel_E & EDR_Eret
```

OpQuad Field FaultPC(31:0)

Field FaultPC[31:0] holds the logical x86 instruction fault program counter value associated with Ops in the OpQuad. In the case of a dual hardware decode, the FaultPC field holds the value of the PC associated with the first of the two instructions. The OCU uses the FaultPC field when handling fault exceptions for any of the Ops in the OpQuad. The logic associated with the FaultPC field, which is a static field, is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

OpQuad Field FaultPC

```
FaultPC = DEC_IPC // the logical PC for the first
                  // decoded x86 instruction in the
                  // OpQuad.
```

OpQuad Field BPTInfo(14:0)

The BPTInfo[14:0] field holds branch prediction table-related information from when the OpQuad was generated. The BPTInfo field is defined only for OpQuads generated by the hardware decoders which contain a BRCOND Op; it is not defined for OpQuads fetched from the OpQuad ROM. The logic associated with field BPTInfo field, which is a static field, is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

OpQuad Field BPTInfo

```
BPTInfo = DEC_BPTInfo  // information from the
                       // current BPT access.
```

OpQuad Field RASPtr(2:0)

The RASPtr[2:0] field points to the top of the return address stack as of when the OpQuad was generated. The RASPtr field is defined for all Opquads but is significant only for OpQuads that contain a BRCOND Op. When a mispredicted BRCOND Op occurs, the RASPtr field is used to restore the decoder's top of Return Address Stack (RAS) pointer to its value as of the mispredicted branch. Note, the K6's RAS is used only for the implementation of x86 RET instructions. A separate one-deep return address stack is implemented in hardware to support one level of OpQuad Sequence subroutine nesting. The logic associated with the RASPtr field, which is a static field, is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

OpQuad Field RASPtr

```
RASPtr = DEC_RASPtr
        // the current return address stack pointer.
```

Historical Comment and Suggested Readings

Return Address Stacks in Microprogrammable Processors

Standard Computer Corporation's MLP-900 was among the early dynamically microprogrammable processors that supported return address stacks for microcode-level subroutines. The MLP-900 return address stack was also used in the processing of interrupts by automatically saving the return address before branching to the control store address where the class of interrupts to be serviced was located. Additionally, the MLP-900 supported a reasonably complete set of micro-operations that used the return address stack in relatively obvious ways, (e.g., branch and enter a subroutine, conditional branch or return, and branch and increment or decrement). See Harold W. Lawson, Jr's. and Burton K. Smith's article "Functional Characteristics of a Multi-Lingual Processor," in the *Preprints of the 3rd Annual Workshop on Microprogramming*, Buffalo, New York, October 12-13, 1970.

OpQuad Field LimViol

The LimViol field indicates that the OpQuad contains the decode of a transfer of control instruction for which a code segment limit violation was detected on the target address. The LimViol field is actually loaded one cycle later than all of the other fields above (i.e., during the first cycle that the new OpQuad is resident and valid within the scheduler). For most rows, field LimViol is static; however, this field can be changed in the first row and therefore the field must be considered a dynamic field. The logic associated with the LimViol field is given in the following pseudo-RTL description. The LimViol field is initialized to zero (LimViol = 1'b0). The description reflects the fact that LimViol is loaded one cycle later:

Pseudo-RTL Description

OpQuad Field LimViol

```
LdLV = LdEntry[0] & ~DEC_OpQSel_E  // a simple flip-
                                   // flop

if (LdLV) LimViol = DEC_LimViol
```

OpQuad Field OpQV

The OpQV field indicates whether the row contains a valid OpQuad. The OpQV field is used by the global control logic when shifting the OpQuads. Invalid OpQuads may be overwritten if an OpQuad lower in the scheduler is held up. The fields in a row containing an invalid OpQuad have the same values as an aborted OpQuad. An OpQuad can become invalid as a result of an abort.

The OpQuad Expansion Logic initially sets the OpQV field to indicate whether the OpQuad loaded into the top of the scheduler is valid. The logic associated with the OpQV field, which is a dynamic field, is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

OpQuad Field OpQV

```
OpQV = (DEC_OpQSel_E ? EDR_OpQV : DEC_OpQV ) &
       ~ExcpAbort & ~(SC_MisPred & ~BrAbort)

Field OpQV can later be cleared after an abort to
invalidate an OpQuad and prevent execution or com-
mitment.

if (SC_Abort) OpQV = 1'b0
```

OpQuad Field FPOP

The FPOP field indicates that the OpQuad contains a floating-point operation (an FpOP). The associated pseudo-RTL description is as follows:

PSEUDO-RTL DESCRIPTION

OpQuad Field FPOP

```
FPOP = DEC_NPPopV  // indicates that Opquad contains
                   // a floating-point op
```

OpQuad Field ILen0(2:0)

In the case of OpQuads produced from dual hardware decodes, the field ILen0 holds the length in bytes of the first of the two decoded x86 instructions. For all other OpQuads, this field is forced to indicate a zero length. The ILen0 field is used to calculate the proper instruction address for faults on the third or fourth Ops within an OpQuad (i.e, as FaultPC + ILen0). The logic associated with the ILen0 field, which is a static field, is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

OpQuad Field ILen0

```
ILen0 = DEC_ILen0  // instruction length of first
                   // short-decoded instruction
```

OpQuad Fields Smc1stAddr, Smc1stPg, Smc2ndAddr, and Smc2ndPg

The fields Smc1stAddr, Smc1stPg, Smc2ndAddr, and Smc2ndPg hold the first and, if there are instructions from more than one cache line in the OpQuad, the second cache-line addresses covered by the instruction bytes of the instruction associated with the OpQuad. These fields are used in the detection of self-modifying code, which consists of writes to any of the bytes of the current instructions. The logic associated with the Smc1stAddr, Smc1stPg, Smc2ndAddr, and Smc2ndPg fields, which are static fields, is given in the following pseudo-RTL description:

PSEUDO-RTL DESCRIPTION

OpQuad Fields Smc1stAddr, Smc1stPg, Smc2ndAddr, & Smc2ndPg

```
Smc1stAddr[11:5] = DEC_Smc1stAddr[11:5]  // page and address from first
Smc1stPg[24:12]  = DEC_Smc1stPg[24:12]   // cache line
Smc2ndAddr[11:5] = DEC_Smc2ndAddr[11:5]  // page and address from second
Smc2ndPg[24:12]  = DEC_Smc2ndPg[24:12]   // cache line
```

THE SCHEDULER PIPELINE

Physically the scheduler is a large storage structure, holding most of the fields of information describing the Ops that are outstanding in the machine at any point in time. This includes both (a) the static state derived from the original Ops as fetched from ROM or generated by the decoders, and (b) the dynamic state resulting from the processing of these Ops. From a control perspective, however, an alternative and perhaps better view of the scheduler is to view it as a pipelined datapath that generates various bits of control information related to the execution of the Ops through their respective execution unit processing pipelines.

The functioning of the scheduler pipeline and the generation of the appropriate control signals are based on the State field associated with each scheduler entry. As we saw in the section titled “Dynamic Field State[3:0]” beginning on page 205, the bits in an entry’s State field reflect the progress of that Op through the appropriate execution unit pipeline.

All changes to the processor state are synchronous with the clock in a very strict manner, (i.e., all state changes effectively occur on the rising edge of the clock). This means, among other things, that processor state changes do not occur in the middle of a cycle and multiple changes to a given element of state do not occur within one cycle or, if they do, the one that occurs on the rising edge of the clock will take effect. In essence, the scheduler, at this level of abstraction, can be thought to be comprised of edge-triggered flip-flops for all storage elements that store processor state information.

The pipelined nature of the processing of Ops is reflected in the structure of the scheduler itself. The overall scheduler and, correspondingly, each Op entry can be divided into many distinct, independent portions (chunks) of logic, each of which is directly associated with a specific processing pipeline stage of a given type of Op or execution pipeline. Correspondingly, from the perspective of a particular processing pipeline, there is a chunk of scheduler logic associated with each pipeline stage that provides key control information for the processing done in that stage and/or for determining when that stage can successfully complete. From the perspective of a given pipeline stage, as viewed across all processing pipelines (at least for the first few common pipeline stages), there are n sets of similar chunks of logic that perform the same function for each pipeline or for each Op source operand of each pipeline.

Suggested Review

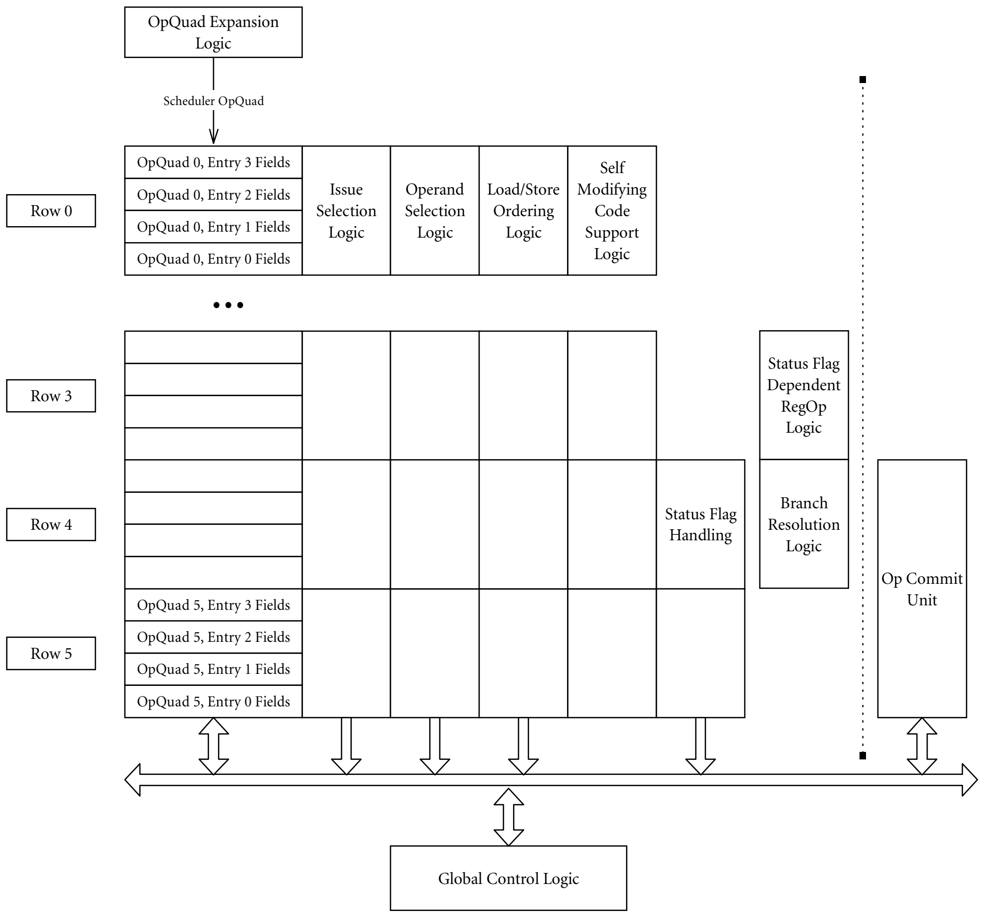
It might be useful at this point for you to review several pipeline diagrams shown in Chapter 2 (Figure 2.12, Figure 2.14, Figure 2.15, Figure 2.18, and Figure 2.21) and the text that accompanies them as well as the scheduler diagram, Figure 2.9, and its related discussion.

Each Op goes through a multistage pipeline as it is processed and transitions between several states:

1. the first two pipeline stages are common to all Ops and represent Op Issue stage and Operand and Operand Transfer stage.
2. the last one or two stages are the actual execute stages.

For integer RegOps there is a single execute stage, corresponding to the fact that all RegOps execute in a single cycle. Further, once an integer RegOp enters this stage, it always successfully completes and exits this stage at the end of the cycle. Some MMX/3D RegOps take two cycles and some can be held up in execute Stage 1. For LdStOps there are two execute stages, during which address calculation, segment and page translation and protection checking, and D-Cache accessing (in the case of LdOps) all take place. Furthermore, LdStOps can be held up for arbitrary periods of time in either stage. Most reasons for holdups apply to the last stage, most notably D-Cache and D-TLB misses, and faults. Holdups in the first of the two execute stages stem typically from misaligned memory references[26] and the second execute stage being occupied and blocked, (i.e., not advancing).

The scheduler has limited involvement with the control of the execute stages of the processing pipelines; it simply keeps track of the state of each Op as it is executed and captures resultant register and status values as and when appropriate. Because of its usefulness at this point, Figure 2.9 is reprinted here:



REPRINTING OF FIGURE 2.9, THE SCHEDULER AND ITS CENTRALIZED BUFFER

[26] Actually there are some miscellaneous additional cases that occur during cache and TLB fill which are not worth describing in any detail. The idea was to give you an example that already appeared in Chapter 2.

The various chunks of scheduler logic shown in this figure, namely the

1. issue selection logic
2. operand selection logic
3. load/store ordering logic
4. status flag handling logic
5. status flag dependent RegOp logic
6. branch resolution logic
7. self-modifying code support logic
8. global control logic

will be summarized and then discussed in more detail using this pipeline framework in the following sections. We will show how the information in the static and dynamic fields in the scheduler is used during the processing of instructions.

An inspection of Figure 2.12, Figure 2.14, Figure 2.15, and Figure 2.18 shows the scheduler processing in the first two common pipeline stages, where each stage consists of two phases. Each phase nominally occurs during the first and second halves of a cycle. These common stages are shown in Figure 3.3.

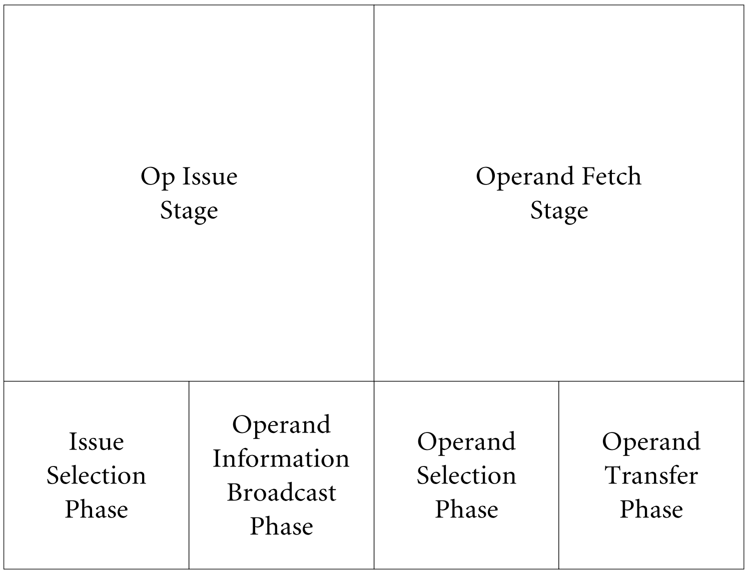


Figure 3.3 Common Pipeline Stages for All RegOps, LdOps, and StOps

The Issue Selection Logic and the Operand Selection Logic shown in Figure 2.9 are directly related to the Op Issue Stage and the Operand Fetch Stage, respectively.

Recall that we introduced some textual abbreviations in Figure 2.16, Figure 2.19, and Figure 2.20 that help reduce the visual clutter of those already crowded figures. Using "S" to stand for stage and "C" to stand for commit, we introduced the notation S_i at the left hand side of Figure 2.16 and Figure 2.19 and the notation C_i in Figure 2.20. We then summarized these pipeline notational correspondences in Table 2.43. We now extend that table to reinforce the fact that neither Figure 2.16 nor Figure 2.19 show the Op Issue Stage.

Table 3.10 ADDITION OF NEW ROW TO TABLE 2.43

Figure 2.16 and Figure 2.19	Figure 2.12, Figure 2.14 and Figure 2.18
Not shown in either figure	Op Issue Stage
S0	Operand Fetch Stage
S1	Execution Stage 1
S2	Execution Stage 2
C	Commit Stage

The use of C1, C2, and C3 in Figure 2.20 reflects the fact that the overall Commit Stage for StOps is composed of several "stages." We will find this useful when discussing the operation of the Store Queue Commit and L1 D-Cache Access logic. The registers shown in Figure 2.16, Figure 2.19, and Figure 2.20 in between the pipeline stages (i.e., S0, S1, S2, C, C1, and C2) are the pipeline registers discussed earlier in this section.

OP ISSUE STAGE LOGIC OVERVIEW

The Issue Selection Stage Logic supports the requirements of both the Op Issue Selection Phase and the Operand Information Broadcast Phase.

Issue Selection Phase

During the Op Issue Selection Phase, the scheduler selects the next Ops to enter the LU, the SU, the RUX, and the RUY processing pipelines—(i.e., four Op selections occur). Each cycle, based on the updated State field of all the scheduler Op entries as of the beginning of the cycle, the scheduler performs a selection process to determine the next LdOp, StOp, and the next two RegOps to be issued into the corresponding execution unit processing pipelines. This will be discussed in detail later in this section.

Operand Information Broadcast Phase

results buses

During the Operand Information Broadcast Phase of the cycle, information is broadcast to all scheduler entries and to external logic about each operand required by the Ops that were selected in the Op Issue Selection Phase. The Operand Information Broadcast Phase sets the scheduler up for actually locating where the appropriate operand values need to come from: (a) a scheduler Op entry, (b) the architectural register file, or (c) the results buses of the execution units (i.e., the *bypass* case). Since each of the four Ops may have up to two operands, a total of eight operand values might be involved. The store data operand for StOps represents a ninth register operand to be fetched, but this operand is not fetched until later in the SU pipeline, (i.e., at the latest possible moment before execution completion).

Operand Fetch Stage Logic Overview

The Operand Selection Logic supports the requirements of both the Operand Selection Phase and the Operand Transfer Phase.

Operand Selection Phase

During the Operand Selection Phase, the scheduler determines:

1. where each of eight operand values actually needs to come from—including which specific scheduler Op entry, architectural register, or execution unit result bus.
2. the status of each value, (i.e., whether a valid value is or is not available from the designated source).

Based on this information, the scheduler determines which of the current Stage0 Ops will be able to advance into Stage1 of their respective execution pipelines. This determination is made independently for each Op. Only explicit operand dependencies constrain the order with which Ops are actually executed. Different types of Ops are processed through their respective execution unit pipelines in arbitrary order with respect to other types of Ops after explicit operand dependencies are taken into account.

Operand Transfer Phase

operand buses

During the Operand Transfer Phase, eight operand values are transferred from the designated sources over the operand buses to the LU, SU, RUX, and RUY execution units. The transfers occur irrespective of whether the values are valid or not. If a value is invalid, then the value will not be used by the execution unit since the associated Op will not have advanced. Once an Op enters pipeline Stage1, the associated execution unit has latched its required operand values and will hold them as long as it remains in Stage1.

displacement buses

Also during the Operand Transfer Phase, two displacement operand values are transferred over the displacement buses to the Load Unit and the Store Unit execution units (one to each unit). The displacements are 32-bit values and always come from scheduler Op entries; in particular, from the DestVal field of the LdStOp's associated scheduler Op entry; see the section titled "The DestVal field plays an important role in the K6's implicit renaming strategy. The OpQuad Expansion Logic circuitry used to initialize the DestVal field and the scheduler circuitry logic associated with dynamic field DestVal is given in the following pseudo-RTL description:" beginning on page 210. The selection of the source entries occurs during the Operand Selection Phase in a relatively trivial manner. When a LdOp or a StOp enters pipeline Stage1, the transferred displacement values are latched by the Load Unit or the Store Unit execution unit along with the associated register operand values.

Immediate values, which can exist as Src2 operands of RegOps, are handled as part of the operand transfer mechanism. In such cases, forwarding of a register value is inhibited and the immediate value is forwarded in its place, using the operand buses, directly from the DestVal field of the scheduler entry holding the Op requiring the immediate value.

Design Note

Store Data Register Value for a StOp

In addition to the above process for obtaining the source operands of the next Ops that will start execution, a similar process is performed for obtaining the store data register value for a StOp. The process is virtually identical to that just described. The only difference is that the four phases—the Issue Selection Phase, the Operand Broadcast Phase, the Operand Selection Phase, and the Operand Transfer Phase—occur in synchronization with pipeline Stage1 and pipeline Stage2 of the StOp. In this case, issue selection is interpreted as the trivial selection of the StOp currently in the SU pipeline Stage1. In essence, store data is fetched in parallel with StOp execution. The actual data value is obtained concurrent with completion of StOp execution. If a valid value is not available yet, then the StOp is held up in Stage2. When a StOp successfully completes execution and exits Stage2, this data and the associated store address is put into the Store Queue as part of creating a new Store Queue entry for this StOp. Effectively, the Store Queue entry is created at the end of pipeline Stage2 and exists as a valid entry starting with the next clock cycle. We will revisit this topic later.

LdOp-StOp Ordering Logic Overview

There are two chunks of logic, one associated with the LdOp pipeline and one associated with the StOp pipeline, that comprise the LdOp-StOp Ordering Logic. Just as certain execution ordering must be maintained between Ops due to register dependencies, a certain degree of execution ordering must be maintained between LdOps and StOps due to memory dependencies. For example, LdOps cannot freely execute ahead of older StOps.

Only a relatively limited amount of ordering is maintained between LdOps and StOps and it is enforced only at Stage2 of the two execution pipelines. This occurs in the form of holding up a LdOp or a StOp in Stage2 until it is acceptable or safe to allow the Op to complete. Up until this point, no ordering is maintained between the two processing pipelines. Further, LdStOps are generally allowed to complete out of order when memory independence can be proved based on partial address comparisons with older LdStOps that are somewhere in the other pipeline, (i.e., when the least significant, untranslated physical address bits are reliably available for such a comparison).

The word "safe" in the above paragraph means the LdOp and StOp in question access a disjoint set of memory bytes. The word "independent" is used as a synonym in this context. Given the splitting of misaligned StOps into pairs of StOps, this can be determined simply by the comparing of 28-bit octet addresses and the 8-bit byte marks. The Ops in question are independent if any part of the address bits don't match or if the two sets of byte marks and non-overlapping (i.e., the bit-wise AND of the two sets of byte marks equals all zeros). Clearly a comparison of a subset of address bits can be sufficient to show two LdStOps are independent; however, all address bits may need to be examined to prove either independence or dependence. Although dependence checks between LU Stage2 and SU Stage2 use full-address and byte mark comparisons, dependency checks between Stage2 and Stage1 pipeline stages use partial-address and full-byte-mark comparisons to determine independencies in the majority of cases (statistically speaking).

The actual address comparisons associated with dependency checking are external to the scheduler, within the LU and SU. However, scheduler support is required for determining the relative age of the LdOps and StOps in the Load Unit and Store Unit execution pipelines. This is necessary so that only the appropriate address comparisons are considered in determining whether a given LdOp or StOp can be allowed to complete. The two chunks of logic referred to are used in these comparisons. The chunk of logic associated with the LdOp pipeline Stage2 determines the age of any LdOp in that stage with respect to any StOps in the StOp pipeline Stage1 or Stage2 and any other StOps which are in earlier stages of processing. The chunk of logic associated with the StOp pipeline Stage2 determines the age of any StOp in that stage with respect to any LdOps in the LdOp pipeline Stage2 and any other LdOps which are in earlier stages of processing.

In the case of a LdOp in LU Stage 2 and older StOps before SU Stage1 (i.e., not having started execution yet), not even a partial-address comparison can be performed. In such cases, which statistically are not performance critical, the temporary conservative assumption is made that these Ops may be dependent and consequently the LdOp is held up in LU Stage2 until a better dependency check can be performed. Similarly, in the case of a StOp in SU Stage2 and older LdOps before LU Stage2, the StOp is *blindly* held up in Stage2. Address comparison with LU Stage1 is not supported since this has minimal performance benefit and saves LdOp-StOp ordering logic in the scheduler.

Status Flag Handling Logic Overview

Lastly, in addition to the chunks of scheduler logic associated with Op issue, register operand fetch, and LdOp-StOp ordering, there is a chunk of scheduler logic associated with the fetching and usage of status flag values. Three relatively independent areas are involved: the fetching of status flag values for status-dependent RegOps, the fetching of status flag values for the resolution of BRCOND Ops, and the synchronization of nonabortable RegOps with surrounding Ops. There is no *explicit* synchronization with BRCOND Ops per se. Instead, this synchronization roadway simply enforces that nonabortable RegOps will always execute in OpQuad[4] of the scheduler and not above or below OpQuad[4]. This, combined with appropriate OpQuad Sequence coding rules and the fact that nonabortable RegOps occur only in OpQuad Sequences, allows the OpQuad Sequence programmer to efficiently achieve the necessary synchronization or serialization with any surrounding dependent or potentially affected Ops (both preceding as well as following).

Status Flag Dependent RegOp Logic Overview

cc-dependent Ops
cc-dep Ops

All status-dependent RegOps, which are referred to as condition code dependent, “cc-dependent,” or “cc-dep” Ops, are executed only by RUX and require their status operand value with the same timing as their register operand values, (i.e., by the end of pipeline Stage0). Unlike the fetching of register values, though, the entire status fetch process is not pipelined and occurs in one cycle, (i.e., entirely during RUX pipeline Stage0). Further, a common set of logic serves to fetch appropriate up-to-date status flag values for both cc-dependent RegOps and for resolution of BRCOND Ops. In the former case these values are simply passed on to the RUX execution unit while validity of the status values needed by the RegOp is checked. If valid values for the required status flags are not yet all available, then the RegOp is held up in pipeline Stage0, just as is done for register operand values not yet available when needed.

Branch Resolution Logic Overview

As is seen from Figure 2.21 on page 175, BRCOND Ops do not require any actual execution processing. Instead, while a BRCOND Op is outstanding and before it reaches the bottom row of the scheduler's buffer, it must be resolved as to whether it was correctly predicted or not. This is done for each BRCOND Op as it passes through OpQuad4 of the scheduler, in order, at a rate of up to one per cycle. When the above status fetch logic obtains the appropriate status for the next unresolved BRCOND Op, the set of status flag values is passed to condition code evaluation logic which determines whether the condition code specified within the BRCOND Op is TRUE or FALSE. If valid values for the required status flags are not yet all available, then resolution of the Op is held up.

If the branch condition is FALSE, the BRCOND Op was incorrectly predicted. In this case, the BRCOND Op is marked as mispredicted and the appropriate restart signal is asserted to restart the upper portion of the processor at the correct branch address (see Figure 2.4 on page 87). Only the upper portion is restarted at this point. The scheduler and the rest of the lower portion are not flushed and restarted until the branch abort cycle. The correct branch direction is the alternative or not-predicted branch direction and the resulting address may be either the sequential or the target address. If the branch was correctly predicted, then nothing happens other than the BRCOND Op is marked as predicted correctly and BRCOND resolution processing advances on to the next BRCOND Op.

This treatment for handling BRCOND Ops applies to BRCOND Ops from the hardware decode of x86 conditional branch instructions and from within OpQuad Sequences. The only difference is whether the alternative address is an x86 instruction address or an OpQuad Sequence address.

The State of BRCOND Ops is initially set to *unissued* (0000). When the Op is resolved, it is left in this state if it is mispredicted or it is changed to *complete* (1111) if it is predicted correctly. Later, based on the state of the Op, the OCU knows whether a branch abort cycle is necessary or not. Typically, RegOps and BRCOND OPs are "trivially" handled by OpQuad Sequences avoiding placing two such Ops in the same OpQuad. As a result, only one of these executes or resolves in any given clock. These executions/resolutions are naturally strictly ordered in OpQuad Sequence programming order. Further, a nonabortable RegOp in the next OpQuad after a BRCOND Op is kept from executing if the BRCOND Op was mispredicted, by the SC_MisPred flip-flop being set (i.e., by that signal being asserted).The case of a nonabortable RegOp and a BRCOND Op in the same OpQuad is allowed (and sometimes actually done) when the

BRCOND is status-flag-dependent on the nonabortable RegOp (since this ensures that the BRCOND Op is not resolved and the SC_MisPred flip-flop possibly set until after the nonabortable RegOp has executed.

Global Control Logic Overview

Basically, the global control logic coordinates the overall operation of the scheduler. For example, among other things it is involved with: RegOp Bumping, the control of the source operand input multiplexers for each of the execution units, validity of each operand value being transferred, and generation of the pipeline advance signals that enable the pipeline registers for each pipeline stage.

Self-Modifying Code Support Logic Overview

Logically, in the scheduler, a detection of self-modifying code is treated as a “kind” of trap of the modifying instruction and factors into the K6’s “trap pending” logic. The self-modifying code support logic detects the existence of such situations and deals appropriately with them.

Issue Selection Logic

As mentioned above, each cycle, based on the updated State fields of all the scheduler Op entries as of the beginning of the cycle, the scheduler performs a selection process to determine the next LdOp, StOp, and the next two RegOps to be issued into the corresponding execution unit processing pipelines. This selection is based solely on the State field and the Type field within each Op entry and essentially results in an in-order issue selection to each type of execution pipeline. The selection is not based on any consideration of the register, status, or memory dependencies that each Op may have on older Ops since such dependencies are not yet known (or at least not within the context of a reasonably short clock cycle time).

This selection process is physically performed simultaneously and independently for each of the four pipelines. The selection algorithms for all four pipelines are similar—the next unissued Op, as indicated by its State field, of the given type of Op is chosen. In other words, the next unissued LdOp is selected for the Load Unit, the next unissued StOp is selected for the Store Unit, and the next two unissued RegOps are selected for RUX and RUY, the first to RUX and the second to RUY. Conceptually, as described earlier in this chapter, the issue selection for RUY is dependent on RUX, but physically is performed in parallel with RUX issue selection. Also as was discussed earlier, some RegOps are only issueable to RUX and thus the Op selected for issue to RUY is the next RegOp that is in fact issueable to RUY. The following describes the selection algorithm in generic terms:

The Selection Algorithm

Each scheduler Op entry generates a signal that represents whether that Op is currently eligible for issue selection to the appropriate pipeline. These signals,

"Issueable to XXX" =
(State = Unissued) AND ("Executable by XXX")

and their pseudo-RTL description were discussed earlier. The selection process consists of a scan, from the oldest scheduler Op entry to the youngest,[27] to locate the first entry with its "Issueable to XXX" signal asserted. The first such Op located is the one selected for issue to execution unit XXX. For RUY issue selection, it is the first such Op after the Op selected for RUX that is selected. The terms in the "Issueable to XXX" equation use information from the State field and Type fields of an Op entry. Specifically, "State = Unissued" = ~S0. The Type field, as shown in Table 3.4 on page 199, indicates if an Op is "Executable by XXX" where XXX = LU, SU, RUX, RUY for the execution pipelines Load Unit, Store Unit, RUX, and RUY respectively. This process can be shown abstractly in the Figure 3.4:

Ops are eligible for issue selection immediately after being loaded into the scheduler which means that an Op can be issued during its first cycle of residence within the scheduler. In such cases, only the Type field and State[0] need to be valid at the beginning of the cycle. All other bits comprising an Op entry can be generated as late as during the issue selection phase (i.e., up to one half cycle later); they only need to be valid within a scheduler entry and set up for the next phase of the processing pipeline.
If an Op selected for issue does not actually advance into pipeline Stage0, then effectively it was not successfully issued. During the next cycle its state will continue to indicate *unissued*. During the next cycle, it will recompete for issue and will be available to be selected again. In the case of RegOps, it is not guaranteed that the Op will be immediately reissued due to the implementation of "RegOp bumping" which is described in the section titled "RegOp Bumping" beginning on page 254.

[27] That is, from the bottom row of the scheduler to the top row.

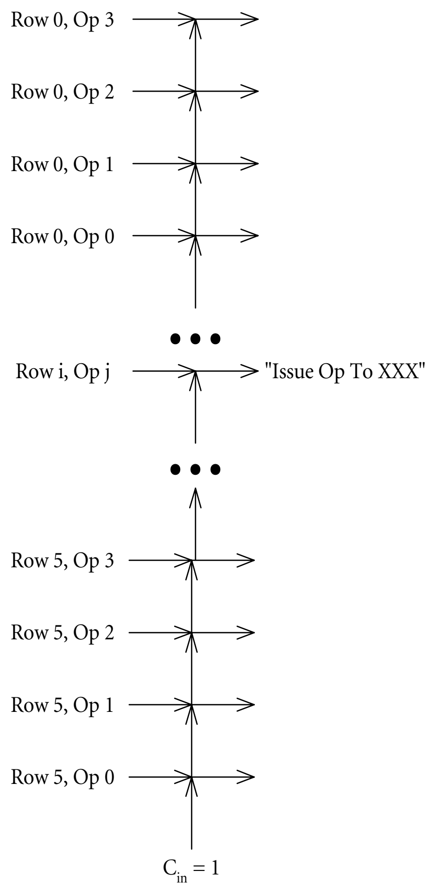


Figure 3.4 Op Issue Selection

Scan Chains

The scheduler's scanning process can be viewed from an implementation perspective as a simple form of carry chain. The carry-in, C_{in},is injected into the beginning of the scan chain at the bottom of the scheduler at the start of the cycle. $C_{in} = 1$ and begins to advance through each bit position either propagating or being *killed*:

Kill = ~Propagate = "Issue To XXX".

The C_{in} to each bit position (i.e., to each Op entry) indicates whether that entry can be selected:

"Selected for issue to XXX" = "Issueable to XXX" & ($C_{in} = 1$)

A "ripple-carry" style implementation of this process is shown in Figure 3.5 below:

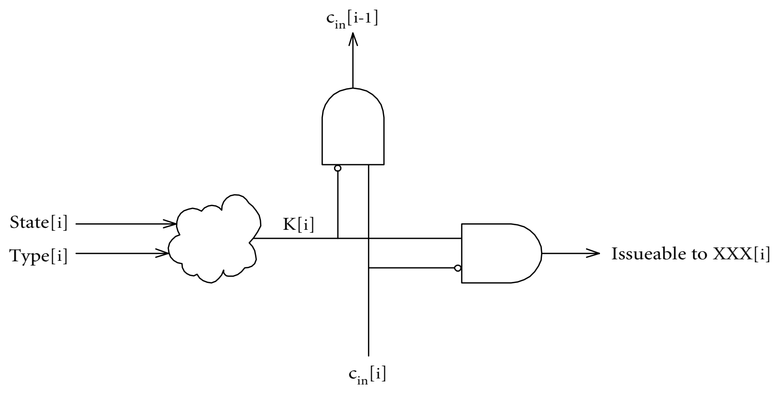


Figure 3.5 SCAN CHAIN STYLE IMPLEMENTATION OF OP SELECTION

lookahead techniques

The final C_{out} from the youngest Op entry is also used by peripheral scheduler logic to indicate whether any Op was found and selected for issue. The scan chain for RUY is more complicated and is discussed below. While a serial scan implementation is very simple, a substantially faster implementation was necessary for the K6 given the central role this scan algorithm plays in issue selection. As with conventional carry chains, carry lookahead techniques can be applied.

generate, propagate, kill signals and equations

The following discussion provides a more concrete description of each scan chain in terms of carry lookahead equations analogous to the traditional Generate-Propagate-Kill equations used in traditional adders. For the LU, SU, and RUX scan chains, the bit-level K terms are defined, the G (generate) terms are all zero, and the P (propagate) terms are the complement of the associated K (kill) terms; i.e., as defined earlier,

$$G[i] = 0, P[i] = \sim K[i].$$

For the RUY scan chain a more complex set of lookahead equations is necessary to do the scan in parallel with the RUX scan, instead of being dependent on it. Four terms are needed: G, P, K, and O. The first three terms are analogous to the conventional terms. At the bit level, the O and G terms are identical and are the same as the bit-level K terms for the RUX scan chain. The K terms are analogous to the K terms for the other chains and the P terms are again the complement of the associated K terms. The overall lookahead equations are extended forms of the conventional ones. The following bit-level G, P, K, and O terms are based on the State and Type fields of an Op entry. The operative definition of the Type field stems from its usage here, i.e. LU = 1 for LdOps, SU = 1 for StOps, RU = 1 for all RegOps, and RUY = 1 for RegOps executable by RUY. These bits are generated by the OpQuad Expansion Logic as Ops are loaded into the scheduler. This is trivially done without any logic since a bit in the RegOp format indicates this.

PSEUDO-RTL DESCRIPTION

Scan Chain Equations

```
Bit-level or Op entry equations
LU: ~P = K = LU ~S0
SU: ~P = K = SU ~S0
RUX: ~P = K = RU ~S0
RUY: ~P = K = RUY ~S0
 O = G = RU ~S0

Group lookahead equations (based on four-bit groups)
LU,SU,RUX: Pgrp = P0 P1 P2 P3
      CIn0 = Cin
           CIn1 = CIn P0
           CIn2 = CIn P0 P1
           CIn3 = CIn P0 P1 P2
           COut = CIn P0 P1 P2 P3
RUY: Pgrp = P0 P1 P2 P3
     Ogrp = O0 + O1 + O2 + O3
     Ggrp = G0 P1 P2 P3 + ~O0 G1 P2 P3 + ~O0 ~O1 G2 P3 +
            ~O0 ~O1 ~O2 G3
CIn0 = CIn
CIn1 = CIn P0       + G0
CIn2 = CIn P0 P1    + G0 P1    + ~O0 G1
CIn3 = CIn P0 P1 P2 + G0 P1 P2 + ~O0 G1 P2 + ~O0 ~O1 G2
COut = CIn P0 P1 P2 P3 + G0 P1 P2 P3 + ~O0 G1 P2 P3
       + ~O0 ~O1 G2 P3 + ~O0 ~O1 ~O2 G3

Issue selection equations
Issue OPi to LU  = LUchain.CINi  LUchain.Ki
Issue OPi to SU  = SUchain.CINi  SUchain.Ki
Issue OPi to RUX = RUXchain.CINi RUXchain.Ki
Issue OPi to RUY = RUYchain.CINi RUYchain.Ki
```

OPERAND INFORMATION BROADCAST

During the Operand (information) Broadcast Phase of the Op Issue Stage of the four processing pipelines, information about each of the four selected Ops is broadcast to all scheduler entries and to external logic. This information describes the two source register operands required by each Op. In addition, other information about each selected Op is also sent to external logic for use later by the associated execution units when they execute the Ops.

operand information buses

In total there are eight operand information buses that run through the scheduler. Each is driven by an issue-selected Op and goes to comparison logic within each Op entry. The total number of comparisons is equal to (8 * the number of Op entries). The results of all of these comparisons are 1-bit signals that control the actions that occur during Operand Selection Phase, which is the next processing phase.

Each operand information bus is eight bits wide and carries the five-bit register number and the three byte marks for a source operand. For each Op that has been selected for issue, the Src1Reg, Src2Reg, Src1BM, Src2BM, and Src12BM fields of its scheduler entry are driven, during the operand broadcast phase, onto the pair of operand information buses corresponding to the processing pipeline that the Op has been selected to be issued to. The following pseudo-RTL description summarizes the equations determining which operand information buses, if any, are driven by a scheduler Op entry during this phase of a cycle:

PSEUDO-RTL DESCRIPTION

Operand Information Bus Equations

```
Src1Info[7:0] = {Src1BM[1:0], Src12BM[2], Src1Reg[4:0]}
Src2Info[7:0] = {Src2BM[1:0], Src12BM[2], Src2Reg[4:0]}

OprndInfo_Lusrc1  = "Issue Op to LU"   ? Src1Info : 8'bZ
OprndInfo_Lusrc2  = "Issue Op to LU"   ? Src2Info : 8'bZ
OprndInfo_Susrc1  = "Issue Op to SU"   ? Src1Info : 8'bZ
OprndInfo_SUsrc2  = "Issue Op to SU"   ? Src2Info : 8'bZ
OprndInfo_RUXsrc1 = "Issue Op to RUX"  ? Src1Info : 8'bZ
OprndInfo_RUXsrc2 = "Issue Op to RUX"  ? Src2Info : 8'bZ
OprndInfo_RUYsrc1 = "Issue Op to RwUY" ? Src1Info : 8'bZ
OprndInfo_RUYsrc2 = "Issue Op to RUY"  ? Src2Info : 8'bZ
```

During the latter part of the operand broadcast phase, every Op entry monitors the operand information buses and checks for matches between its Op's destination register and any of the source operands about to be fetched. This comparison logic checks both for matching register numbers and for overlapping byte marks as some or all of the bytes required for an operand are or will be modified by this Op. The following pseudo-RTL description summarizes the generic comparison equation:

PSEUDO-RTL DESCRIPTION

Destination Register & Source Operand Comparisons

```
match with operand XXsrcY:
OprndMatch_XXsrcY =(bus.Reg[4:0] = DestReg[4:0]) &
                   (bus.BM[1] & DestBM[1] | busBM[0] & DestBM[1])

where XXsrcY takes on the values LUsrc1, LUsrc2, SUsrc1, SUsrc2, RUXsrc1,
RUXsrc2, RUYsrc1, and RUYsrc2, and "bus" refers to OprndInfo_XXsrcY.

The byte mark checking does not include BM[2], as a simplification, since
(BM[2] = 1'b1) implies (BM[1] & BM[0]); thus, if (bus.BM[2] = 1'b1), then a
match will be signaled irrespective of DestBM[2]
```

pipeline registers
match signals

The results of these comparisons represent the output of the operand broadcasting phase and are captured in pipeline registers for use in the Operand Selection Phase of Stage0, which is the very next pipeline stage. This is done concurrently within each and every Op entry—within each entry, eight match signals (the values of the comparison results) are captured in pipeline registers to be used in the entry's operand selection logic. All of the match signals remain local to each Op entry. In essence, within each entry, eight operand information bus comparators feed eight "control" signals (the match signals) to eight chunks of Operand Selection Logic. We will learn later that the match signals within each Op entry in the bottom row of the scheduler are masked by additional signals associated with the committing of the results of the Ops in this OpQuad to the architectural register file.

The loading of the pipeline registers that capture the match signals is not controlled by logic within the entry. Instead, the registers within each entry associated with the two load unit operands are controlled by global control signals generated by scheduler peripheral logic. For example, all pipeline registers within the scheduler associated with the LU pipeline Stage0 are controlled by the global signal LUAdv0. The global control signal is a function of whether an Op selected for issue to the LU can advance into pipeline Stage0. Similarly, the registers across all entries that are associated with the SU, RUX, and RUY Stage0 are controlled by SUAdv0, RUXAdv0, and RUYAdv0,respectively.

Besides the internal use of the values on the operand information buses, these values are also latched into external pipeline registers for use by peripheral logic. Additional information about each selected Op, namely the OpInfo field, is also read out of the scheduler during this phase and latched into external pipeline registers; see the section titled "Static

Field OpInfo[12:0]" beginning on page 204. The following summarizes the equations representing this readout of OpInfo fields:

Pseudo-RTL Description

OpInfo Field Readout

```
OpInfo_LU  = “Issue Op to LU”  ? OpInfo: 13'bZ
OpInfo_SU  = “Issue Op to SU”  ? OpInfo: 13'bZ
OpInfo_RUX = “Issue Op to RUX” ? OpInfo: 13'bZ
OpInfo_RUY = “Issue Op to RUY” ? OpInfo: 13'bZ
```

All of these external pipeline registers are controlled in the same way as the above internal registers, (i.e., by the XXAdv0 signals). The Src1Reg, Src2Reg, Src1BM, Src2BM, and Src12BM fields are used for a number of purposes during the next two pipeline phases, (i.e., during pipeline Stage0). The OpInfo fields are simply passed “down the pipeline” to the corresponding execution units through a second set of pipeline registers controlled by the corresponding XXAdv1 signals.

Operand Selection Logic

Each cycle, based on the values of the match signals in the pipeline Stage0 operand match registers as generated by the above Op issue stage chunk of logic, the scheduler performs a selection process to determine which Op entry, if any, will supply the operand value for each register operand being “fetched”. This is called the Operand Selection Phase. It must also be determined during this phase whether each operand’s value will come from the scheduler or the architectural register file. The architectural register file is the default source for these values if there is no matching Op entry to supply the value. As with the Op selection process in the Op issue stage pipeline, the operand selection process is independently and simultaneously performed for each operand being fetched. Thus there are eight chunks of Operand Selection Logic. The operand selection process is very similar to the Op issue selection process. During the next Stage0 phase, the Operand Transfer Phase, the operand values from the selected scheduler entries or register file will be driven onto the operand buses and transferred to the associated execution units.

For each operand, there is an operand match register bit in each Op entry (see the section titled “Dynamic Fields OprndMatch_XXsrcY” beginning on page 212). There are a total of (8 * “# of Op entries” = 192) such operand match register bits. The selection algorithm is to find the youngest Op entry with a match which is older than the Op entry containing the Op whose operand is being selected (i.e., fetched).

Design Note, Historical Comment, and Suggested Readings

Operand Forwarding and Register Bypassing

During each cycle, the selection process determines what needs to be done to forward appropriate operand values during that cycle in case the Op successfully advances into pipeline Stage1 after this cycle. If an Op, whose operands are being selected, does not advance out of pipeline Stage0, then the selection process will be performed again the next cycle. Since an Op's state and location within the scheduler can change each cycle, the outcome of the new selection may be different from the current cycle's outcome.

The terms *operand forwarding* or *register bypassing* are typically applied to a design that allows a result value produced by an operation to be used at an earlier stage in a pipeline than it would normally be able to be used by forwarding the output directly from the producing unit to the unit that requires the value as a source operand, bypassing an intermediate load of a register from which it would be subsequently read. See, for example, Michael J. Flynn, *Computer Architecture Pipelined and Parallel Processor Design*, Jones and Bartlett Publishers, 1995.

This can be viewed as a scan, from the Op entry containing the Op whose operand is being fetched and in the direction of older entries, for the first entry with its operand match bit set. This Op is selected as the supplier of the required operand value and thus as the driver of the associated operand bus during the operand transfer phase. As mentioned earlier, if no older matching entry exists, then the register file, by default, is selected to supply the operand value.

The scan to find the appropriate source for an operand value can be viewed, like the Op issue selection scans, as a form of carry chain. In this case, though, the scan chains cannot be simple "propagate-kill" chains. In fact, it turns out that an operand selection scan chain is directly analogous to a traditional carry or "generate-propagate-kill" chain. Furthermore, the scan is in the direction of older Ops which is the opposite direction of the Op issue selection scans.

The initial C_{in} into the least significant bit of the chain is set equal to 0 while a carry generate occurs at the Op entry position corresponding to the Op whose operand is being fetched. Carry kills occur at all Op entries with set operand match bits. Carry propagates occur at all other positions. The Op entry with both a set match bit and a $C_{in} = 1$ is the selected entry. The final C_{out} from the oldest Op entry is also used by peripheral scheduler logic to determine if any entry was selected and thus whether the architectural register file should instead be selected—if $C_{out} = 1$, then use the value in the architectural register file. The selected operand value source drives the corresponding operand bus during the operand transfer phase, (i.e., the latter part of the cycle).

As with the issue selection scan chains, a carry-lookahead-based implementation will be necessary for speed. The following pseudo-RTL description provides a concrete definition of the operand selection scan chain in terms of carry-lookahead equations similar to the traditional generate-propagate-kill equations:

Pseudo-RTL Description

Operand Selection Scan Chain

```
Bit-level or Op entry equations:

LUsrc1:  ~P = K = OprndMatch_LUsrc1
          G = LU & ~S1 & S0
LUsrc2:  ~P = K = OprndMatch_LUsrc2
          G = LU & ~S1 & S0

SUsrc1:  ~P = K = OprndMatch_SUsrc1
          G = SU & ~S1 & S0
SUsrc2:  ~P = K = OprndMatch_SUsrc2
          G = SU & ~S1 & S0

RUXsrc1: ~P = K = OprndMatch_RUXsrc1
          G = RU & Exec1 & ~S1 & S0
RUXsrc2: ~P = K = OprndMatch_RUXsrc2
          G = RU & Exec1 & ~S1 & S0 & ~Imm

RUYsrc1: ~P = K = OprndMatch_RUYsrc1
          G = RU & ~Exec1 & ~S1 & S0
RUYsrc2: ~P = K = OprndMatch_RUYsrc2
          G = RU & ~Exec1 & ~S1 & S0 & ~Imm

Group lookahead equations (based on four-bit groups)

Pgrp = P0 & P1 & P2 & P3
Ggrp = G0 & P1 & P2 & P3 | G1 & P2 & P3 | G2 & P3 | G3
CIn0 = CIn
CIn1 = CIn & P0 | G0
CIn2 = CIn & P0 & P1 | G0&P1 | G1
```

continued on the next page ...

Pseudo-RTL Description (cont.)

Operand Selection Scan Chain

```
CIn3 = CIn & P0 & P1 & P2 | G0 & P1 & P2 |
          G1 & P2 | G2
COut = CIn & P0 & P1 & P2 & P3 | G0 & P1 & P2 & P3 |
          G1 & P2 & P3 | G2 & P3 | G3

Operand selection equations:

“Supply Op[i] result value to LUsrc1”  =
     LUsrc1chain.CIN[i] & LUsrc1chain.K[i]
“Supply Op[i] result value to LUsrc2”  =
     LUsrc2chain.CIN[i] & LUsrc2chain.K[i]
“Supply Op[i] result value to SUsrc1” =
     SUsrc1chain.CIN[i] & SUsrc1chain.K[i]
“Supply Op[i] result value to SUsrc2”  =
     SUsrc2chain.CIN[i] & SUsrc2chain.K[i]
“Supply Op[i] result value to RUXsrc1” =
     RUXsrc1chain.CIN[i] & RUXsrc1chain.K[i]
“Supply Op[i] result value to RUXsrc2” =
     RUXsrc2chain.CIN[i] & RUXsrc2chain.K[i]
“Supply Op[i] result value to RUYsrc1” =
     RUYsrc1chain.CIN[i] & RUYsrc1chain.K[i]
“Supply Op[i] result value to RUYsrc2” =
     RUYsrc2chain.CIN[i] & RUYsrc2chain.K[i]
```

Operand Transfer Logic

During the Operand Transfer Phase of Stage0 of the four processing pipelines, appropriate values for each of the eight register source operands being fetched are transmitted over the eight operand buses to input operand registers of the associated execution units. Integer operand values are always 32-bit quantities; some bytes may be undefined and an execution unit may not use all four bytes. Undefined operand bytes will never be used by an execution unit if the processor is functioning properly. As previously described, each operand bus can be driven by any scheduler Op entry or by the register file. Conversely, any Op entry could drive any or all of the buses or none of them.

During the preceding phase (8 * “# of Op entries” = 192) operand selection signals and eight scan chain carry-outs (C_{out}) were generated. The pseudo-RTL description for the Operand Information Bus Equations on page 233 summarizes the equations that determine which operand information buses, if any, are driven by a scheduler Op entry or the register file during this phase of a cycle.

The operand values transferred over the eight operand buses are captured in pipeline registers at the inputs of the four execution units for use in the next pipeline stage, which is the first execution unit pipeline stage, Stage1. The loading of these pipeline registers is controlled by global control logic signals generated by scheduler peripheral logic as was described earlier for the Stage0 pipeline registers. The LU pipeline Stage1 source operand registers are controlled by global signal LUAdv1; similarly, the SU, RUX, and RUY Stage1 operand registers are controlled by SUAdv1, RUXAdv1, and RUYAdv1, respectively. The signals are a function of whether an Op in a pipeline Stage0 can advance into pipeline Stage1.

MMX/3D operands are handled similarly to integer operands and, notably, they use the same logic. To accomplish this, the following modifications are utilized:

1. the 64-bit MMX/3D registers are treated like 4-byte reads and writes of integer registers insofar as source and destination byte marks, (i.e., all three byte marks are set for MMX/3D registers).
2. separate 5-bit register numbers are used for MMX/3D registers (twenty-four numbers are used for integer registers and eight numbers are used for MMX/3D registers). Consequently, register number comparisons between integer and MMX/3D registers never match, (i.e., operand match and selection logic equations readily handle, without change, both integer operands for integer Ops and MMX/3D operands for MMX/3D-related Ops).
3. copies of the integer operand selection signals are also used as MMX/3D operand selection signals. Similarly, copies of integer operand bypass control signals are also used as MMX/3D operand bypass control signals.

operand status bus

During the operand transfer phase of Stage0, information about each of the Ops selected to provide an operand value is also read out of the scheduler. Each operand bus can be viewed as having an associated operand status bus that carries information describing the "origins" of the value being fetched. This information is used during this phase, by external logic, to determine the availability of a valid operand value and where the operand came from.

Each operand information bus is ten bits wide and contains a number of fields. The following summarizes these fields as taken directly from or derived from fields of the Op entry that is providing the operand value:

OprndStat[9:0] = {State[3:0], DestBM[2:0], Type[2:1], Exec1}

The following pseudo-RTL description summarizes the equations describing this readout of OprndStat information. These equations determine how and when the operand status buses, if any, are driven by a scheduler Op entry during this phase of a cycle. It can be easily seen that they directly correspond to the operand bus driver enable equations just given.

PSEUDO-RTL DESCRIPTION

OprndStat Information

```
OprndStat_LUsrc1 =
  “Supply Op result value to LUsrc1” ? OprndStat : 10'bZ
OprndStat_LUsrc2  =
  “Supply Op result value to LUsrc2” ? OprndStat : 10'bZ
OprndStat_SUsrc1  =
  “Supply Op result value to SUsrc1" ? OprndStat : 10'bZ
OprndStat_SUsrc2  =
  “Supply Op result value to SUsrc2” ? OprndStat : 10'bZ
OprndStat_RUXsrc1 =
  “Supply Op result value to RUXsrc1” ? OprndStat : 10'bZ
OprndStat_RUXsrc2 =
  “Supply Op result value to RUXsrc2” ? OprndStat : 10'bZ
OprndStat_RUYsrc1 =
  “Supply Op result value to RUYsrc1” ? OprndStat : 10'bZ
OprndStat_RUYsrc2 =
  “Supply Op result value to RUYsrc2” ? OprndStat : 10'bZ
```

Just as with the operand buses, there is a similar set of drivers at the output of the register file. As default drivers of these buses, they ensure that the operand status buses always carry defined values and, in these cases, that the values result in appropriate behavior by the external logic using this information. The pseudo RTL-description for OprndStat Information on this page summarizes the equations describing how and when the operand status buses are driven by the register file.

Completion Of Operand Transfer

The source operand value that is eventually delivered to an execution unit can come from any one of three possible sources:

1. a scheduler Op entry.
2. the architectural register file.
3. the result bus coming from the output of this or another execution unit.

The first case is covered by the Operand Transfer Logic described in the section titled "Operand Transfer Logic" beginning on page 238. In the second case, the register file is accessed during the operand selection phase of Stage0. As we learned, the register number of the desired architectural register is read out of the associated Op entry during the operand information broadcast phase of the Op Issue Stage and passed to the appropriate read port of the register file. For both of these cases, the resultant source operand value is transferred, during the Operand Transfer Phase, to the execution unit's operand input via a dedicated operand bus (Oprnd_XXsrcY) where it is multiplexed into the unit's operand register. The selection of whether a scheduler Op entry or the register file is enabled to drive the operand bus is determined by the scheduler during the preceding operand selection phase, as described at the end of the preceding section. The operand registers are controlled/enabled by the XXAdv1 global pipeline control signals. There are five-to-one (5:1) multiplexers which are used to select between the operand bus from the scheduler and the four buses from the LU, SU, RUX, and RUY execution units.

result buses

In the third case above, the value on the operand bus is ignored by the execution unit and, instead, the value on the appropriate result bus (Result_XX) is directly multiplexed into the unit's operand register. Thus, five operand buses run to each operand input of each execution unit, namely the operand bus for that operand input plus the four execution unit result buses. Since there are a total of eight operand fetches per cycle, there are twelve (8 + 4 = 12) buses carrying register values to and from the execution units. There is one additional operand bus, described later, for the store data operand of StOps, see the section titled "Store Data Operands" beginning on page 244.

The control for the 5:1 multiplexer at operand input of each execution unit is generated by the scheduler during the operand transfer phase of Stage0. The scheduler determines if the desired operand value is or may just be coming available, in which case the appropriate result bus is bypassed into the execution unit; otherwise, the operand bus is selected as the input. The validity of the desired operand value is an independent issue that only affects whether the associated Op in pipeline Stage0 will be allowed to advance into pipeline Stage1 and thus actually enter the execution unit.

Displacement Operand & Immediate Value Forwarding

We will consider the logic associated with displacement operands and the logic associated with immediate values separately as there are important differences between them.

Displacement Operand Forwarding

During the operand transfer phase of the Load Unit and the Store Unit processing pipelines, in addition to the register operands for each of these units, displacement operands are fetched and forwarded. The Load Unit and the Store Unit each have three operand buses carrying two register operands and one displacement operand. Displacement operands are always 32-bit quantities that are sign-extended from 8-bit quantities, as need be, when loaded into the scheduler from the decoders.

displacement buses

Displacement values are handled within the scheduler in a manner similar to Op register result values. Until they are used, displacement values are stored within the 32-bit DestVal fields of Op entries. They are driven onto displacement buses during the operand transfer phase of Stage1 of the LU and the SU pipelines. Displacement values are always supplied from scheduler Op entries. They are never forwarded from the register file. This usage of the DestVal field of a LdStOp does not conflict with its normal usage by LdOps and some StOps since result values are not loaded into the scheduler Op entry until after the displacement value is used.

As noted above, all displacements are stored in the DestVal field. Small (8-bit) displacements are expanded during Op decode. The selection between this expanded displacement and the 32-bit dedicated displacement bus occurs during Op decode and the final, effective 16/32-bit displacement value is loaded into the Op's DestVal field.

The selection of DestVal values to drive onto the displacement buses during each cycle does not require a scanning process across scheduler Op entries. Instead, each Op entry enables the drivers of its DestVal field onto the appropriate displacement bus based on its State and Type bits. The following pseudo-RTL description summarizes the equations enabling the displacement bus drivers within each Op entry:

PSEUDO-RTL DESCRIPTION

Displacement Value Selection

```
Disp_LU  = (LU & ~S1 & S0 & ~LD) ? DestVal : 32'bZ
Disp_SU  = (SU & ~S1 & S0 & ~LD) ? DestVal : 32'bZ
```

Immediate Values Forwarding

Immediate values can only exist as Src2 operands of RegOps. They are handled in a manner similar to displacements but as part of the operand transfer mechanism. Like displacement values, they are stored in the DestVal fields of Op entries; like register operands they are forwarded over register operand buses, specifically the RUXsrc2 and RUYsrc2 operand buses.

Only small (8-bit) immediate values need to be handled and are treated just like small displacement values in that they are stored in the DestVal field of the Op entry holding the RegOp using the immediate value. The are stored after suitable sign extension by OpQuad Expansion Logic before being loaded into the DestVal field. Comments regarding storage of displacement values in DestVal fields also apply to storage of immediate values.

Src2 operand immediate values are forwarded to respective RegOp execution units during the operand transfer phase of pipeline Stage0 in place of any register value. The selection of any register value source (a scheduler Op entry or the architectural register file) is inhibited and the RegOp in question directly drives its DestVal field onto the appropriate Src2 operand bus.

The inhibition of any RUXsrc2 or RUYsrc2 operand selection is done during the operand selection phase through masking of the generate signal that would normally be asserted by the Op entry holding the RegOp whose operands are being fetched. This is done separately and independently for RUXsrc2 and RUYsrc2. This prevents selection of any Op entry by the RUXsrc2 and RUYsrc2 scan chain and selection of the register file as the default operand source and is reflected in the previous operand selection scan chain equations.

The selection of immediate DestVal values to drive onto the RUXsrc2 and RUYsrc2 operand buses during each cycle does not require a scanning process across scheduler Op entries. Instead, each Op entry enables the drivers of its DestVal field onto the appropriate operand bus simply based on its State and related bits. These are the same drivers that are used for normal register value forwarding; there is simply an additional term in each enable equation for handling immediate operands. The following pseudo-RTL description summarizes these terms as separate equations enabling separate bus drivers within each Op entry:

PSEUDO-RTL DESCRIPTION

Immediate Value Selection

```
Oprnd_RUXsrc2 =  (RU &  Exec1 & ~S1 & S0 & Imm) ? DestVal : 32'bZ
Oprnd_RUYsrc2 =  (RU & ~Exec1 & ~S1 & S0 & Imm) ? DestVal : 32'bZ
```

When an Op entry drives an immediate DestVal onto an operand bus, it must also drive the associated operand status bus. This is handled in the same manner as it is with the operand buses, (i.e., the same bus drivers and driver input values as for normal operands are used for immediate values). There is simply an additional term in each enable equation—the same additional terms as given above in the immediate value selection equations.

The following summarizes these terms as separate equations enabling separate bus drivers:

PSEUDO-RTL DESCRIPTION

OprndStat Information for Immediate Values

```
OprndStat_RUXsrc2 = (RU &  Exec1 & ~S1 & S0 & Imm) ? OprndStat : 10'bZ
OprndStat_RUYsrc2 = (RU & ~Exec1 & ~S1 & S0 & Imm) ? OprndStat : 10'bZ
```

Store Data Operands

StOps are rather special in that they have three register source operands and (typically) no register destination. This is in contrast to all other Ops, which have up to two register source operands and one register destination. A STUPD Op is an exception insofar as it also has a destination. The data is only needed for completion of the Op. Certain StOps, such as STUPD Ops and LEA Ops, are an exception insofar as they also have a register destination. For LEA Ops there is no store data operand.

As a result, the fetching of StOp data operand values is performed in a manner similar to that for all other register source operands, but it is synchronized with the Store Unit pipeline Stage2 (see Figure 2.18 on page 167 and Figure 2.19 on page 168). Whereas the "normal" operand fetch process occurs during the Op Issue and Stage0 stages of processing for an Op, the store data fetch process occurs during Store Unit pipeline Stage1 and Stage2. If a data value is not yet available, this is realized during Store Unit Stage2 and the associated StOp is held up there.

Given that the data fetch process is largely the same as described in previous sections, the following section simply describes the two principal differences and summarizes all of the corresponding equations that support this process. Thus, this section describes logic within each scheduler Op entry.

The first difference is that the selection of an Op to issue is more simply the selection of the StOp currently in the Store Unit pipeline Stage1. A scan across scheduler entries to choose between multiple selection candidates is unnecessary. The second difference is that the OpInfo field of the StOp does not need to be read out during the operand broadcast phase. Instead, the OpInfo value read out when the StOp was issued is retained and used during the following two data fetch phases. The OpInfo value read out during the Store Unit Op Issue Stage is essentially passed down the Store Unit pipeline through Stage0, Stage1, and Stage 2.

.

PSEUDO-RTL DESCRIPTION

Store Data Operand Fetching

```
Store Unit stage 1: Op Selection

“Select for store data fetch” = SU & ~S2 & S1

Store Unit stage 1: Operand Info Broadcast

SrcStInfo[7:0] = {SrcStBM[2..0],SrcStReg[4..0]}

OprndInfo_SUsrcSt = “Select for store data fetch” ? SrcStInfo : 8'bZ

“match with operand SUsrcSt’:
  OprndMatch_SUsrcSt = (busReg[4:0] = DestReg[4:0]) &
                       (busBM[1] & DestBM[1] | busBM[0] & DestBM[1])

  where “bus” refers to OprndInfo_SUsrcSt.

This match signal is then latched into a pipeline register bit within each
Op entry:

if (SUAdv2) OprndMatch_SUsrcSt = “match with operand SUsrcSt”

Store Unit stage 2: Operand Selection

Bit-level scan equations:
  ~P = K = OprndMatch_SUsrcSt
   G = SU & ~S3 & S2

Group-level scan equations:
  same as for other operand selection scan chains

“Supply OPi result value to SUsrcSt” =
  SUsrcStchain.CIN[i] & SUsrcStchain.K[i]
```

continued on next page ...

Pseudo-RTL Description (cont.)

Store Data Operand Fetching

```
Store Unit stage 2: Operand Transfer

Enable for driver within each Sched Op entry:
  Oprnd_SUsrcSt =
    “Supply Op result value to SUsrcSt'? DestVal : 32'bZ
  OprndStat_SUsrcSt =
    “Supply Op result value to SUsrcSt” ? OprndStat:10'bZ

Enable for driver at output of register file:
  Oprnd_SUsrcSt =
    SUsrcStchain.COUT ? SUsrcStRewgVal : 32'bZ
  OprndStat_SUsrcSt =
    SUsrcStchain.COUT ? {7'b1111111,3'bxxx} : 10'bZ
```

store data operand bus, Oprnd_SUsrcSt bus

The store data operand value transferred over the source data operand bus (the Oprnd_SUsrcSt bus) is captured in a pipeline register at the input of the Store Queue. The operand status value read out is used by external control logic during this phase. There is the same sort of 5:1 multiplexer at the input of the Store Queue like those for the other operand inputs to the execution units.

RegOp Bumping

When Ops are issued to a given execution unit processing pipeline, they typically progress down it in order with respect to other Ops issued to that pipeline. When an Op is held up in pipeline Stage0 the Op currently being selected for issue to that pipeline also gets held up. The scheduler generally manages the execution unit processing pipelines based on in-order issue selection and processing.

When a RegOp is bumped out of either the RUX or RUY pipeline Stage0, the following RegOp selected for issue to that register execution unit advances into Stage0, immediately taking the place of the bumped RegOp. This allows the issue-selected RegOp to “pass by” without delay. Simultaneously, the bumped RegOp is immediately eligible for reselection and Op issue.

DESIGN NOTE

RegOp Bumping

Although Ops can readily pass each other when they are in different processing pipelines, including the two RegOp pipelines, Ops cannot pass each other within any given processing pipeline. One exception is made to this general rule. When a RegOp is held up in pipeline Stage0 of either RUX or RUY due to one or more unavailable operand values, it may be both acceptable (insofar as not causing unnecessary Op execution delays) and desirable (insofar as having positive performance benefit) to bump the RegOp out of its processing pipeline. This *RegOp bumping* is accomplished by clearing the Stage0 valid bit for the RegOp and by resetting its state in the associated scheduler Op entry to *unissued*.

The bumping of a stalled RegOp is generally applicable to all RegOps and is subject only to the following two constraints:

1. a RUX-only RegOp cannot be bumped if a RUX-only RegOp is currently being selected for issue to RUX. Bumping the stalled RegOp would violate the principle that RUX-only RegOps are guaranteed to execute in order with respect to each other. This fact is taken advantage of in many OpQuad Sequences to accomplish various serialization and synchronization purposes (e.g., for multiply, divide, and segment register loading instructions).
2. a RegOp should only be bumped if it is guaranteed to be stalled for more than one cycle, otherwise it is generally better to leave the RegOp in pipeline Stage0 waiting to advance in Stage1. This avoids the additional execution delay and thus detrimental performance impact that could result from bumping a RegOp that only needed to wait one clock before being able to advance into Stage1 and start execution.

The implementation of RegOp bumping is reflected in the equations controlling the S1 State bit of an Op's scheduler entry; see the section titled "Dynamic Field State[3:0]" beginning on page 205. The implementation is also reflected in additional peripheral control logic that generates the global bump signals BumpRUX and BumpRUY and forces assertion of the RUXAdv0 and RUYAdv0 signals.

BumpRUX
BumpRUY

Pseudo-RTL Description

BumpRUX/Y Equations

```
// Inhibit Bumping of RUX
  InhBumpRUX =
    OpInfo_RUX_0.RegOp.R1 & OpV_RUX_Iss & OpInfo_RUX.RegOp.R1

// RUX Time Out
  RUXTimeOut = (RUXTimeOutCnt[2:0] == 0x0)

// Bump RUX Stage0 Op
  BumpRUX =
    (!OprndStat_RUXSrc1.State[0] ||
     !OprndStat_RUXSrc1.State[1] && !OprndStat_RUXSrc1.Type[1] ||
     !OprndStat_RUXSrc2.State[0] ||
     !OprndStat_RUXSrc2.State[1] && !OprndStat_RUXSrc2.Type[1] ||
     RUXTimeOut) &&
    !InhBumpRUX && !RegOpBumpDisable

// RUY TimeOut
  RUYTimeOut = (RUYTimeOutCnt[2:0] == 0x0)

// Bump RUY Stage0 Op
  BumpRUY =
    (!OprndStat_RUYSrc1.State[0]  ||
     !OprndStat_RUYSrc1.State[1]  && !OprndStat_RUYSrc1.Type[1]  ||
     !OprndStat_RUYSrc2.State[0]  ||
     !OprndStat_RUYSrc2.State[1]  && !OprndStat_RUYSrc2.Type[1]  ||
     RUYTimeOut) &&
    !RegOpBumpDisable
```

Load/Store Ordering Logic

In this subsection, we will deal with two important issues: (1) the determination of the relative order or age, within the scheduler, between certain LdOps and certain StOps; (2) the process of maintaining proper execution ordering between dependent LdOps and StOps.

LdOp-StOp Ordering Determination and Control Logic

The scheduler provides key support for maintaining a sufficient degree of execution ordering between LdOps and StOps. This is necessary so that related memory-dependency handling logic can ensure appropriate forwarding of memory data from memory writes to memory reads. As described earlier, only a limited amount of ordering is maintained and is done so only at Stage2 of the Load Unit and the Store Unit execution pipelines. Further, this only applies to StOps which actually reference memory or at least generate fault addresses (i.e., CIA and CDA StOps), although these latter Ops are included only for design simplification reasons. There is no execution-ordering constraint on or with respect to LEA Ops. No LdOps are excluded since all LdOps reference memory.

relative age of LdStOps

The actual LdOp-StOp ordering control logic is based on partial address comparisons between LdStOps in pipeline Stage1 and Stage2 of their respective pipelines. The scheduler provides information about the relative age of such LdStOps so that only the appropriate comparisons are considered in determining whether to hold up a pipeline Stage2 LdOp or StOp. These comparisons are separately performed for any LdOp in Load Unit pipeline Stage2 and for any StOp in the Store Unit pipeline Stage2.

1. in the former case, the scheduler determines the age of any Stage2 LdOp with respect to any StOps in Store Unit Stage1 or Stage2, and any other StOps that are in earlier stages of processing. The purpose in this case is to prevent LdOps from completing execution ahead of Store Queue entries being created for older StOps that the LdOp is or may be dependent on.
2. in the latter case, the scheduler determines the age of any Stage2 StOp with respect to any LdOps in Load Unit pipeline Stage2 and any other LdOps that are in earlier stages of processing. The purpose in this case is to prevent StOps from completing execution and creating Store Queue entries ahead of older LdOps that would eventually look up in the Store Queue and falsely appear dependent on these StOps.

The Relative Age Determination Process

Thus the two cases are not symmetric. The actual relative age determination process is similar to the Op issue selection and operand information broadcast process described earlier and involves the use of scan chains. During the first phase of pipeline Stage2 for a LdStOp, propagate-kill style scans (three scans for LdOps and two for StOps) are performed across all the scheduler Op entries from the oldest to the youngest. Each of the scan chains has its initial carry-in signal (C_{in}) set to 1. The initial Cin is the Cin injected into the beginning of the scan chain. Carries are "killed" by certain LdStOps which are different for each scan chain. During the second phase of Stage2 the C_{in} of the Stage2 LdStOp is selected or read out via a multiplexer, effectively broadcasting its C_{in} values out to peripheral scheduler logic. The value of these signals indicates the desired relative age information.

In the case of either a Stage2 LdOp or a misaligned Stage1 LdOp (for which the first half then is in Stage2 while the second half is in Stage1[28]), three scan

[28] See the section titled "LdOp Misaligned Accesses" beginning on page 171.

chains are needed since their age relative to the following three categories of StOps must be determined:

1. any Stage2 StOp or Stage1 StOp performing the second half of a misaligned StOp.
2. any Stage1 StOp.
3. any pre-Stage1 StOps.

Each scan chain handles one of these cases and each scans for the oldest StOp with a given state. The value of the carry at any point in the scan chain reflects whether a StOp of given state has been found yet. Thus, the LdOp in question can determine its relative age to any StOps in a given category simply by examining the value of the corresponding scan chain C_{in} to its Op entry. If the carry has not been killed yet (i.e., $C_{in} = 1$), then no older StOp of the given state exists. Based on these indications, LdOp ordering control logic can determine which Store Unit pipeline Stage1 or Stage2 address comparators to examine and then whether to hold up the Stage2 LdOp in question.

In the case of either a Stage2 StOp or a misaligned Stage1 StOp (for which the first half then is in Stage 2 while the second half is in Stage 1[29]) performing the second half of a misaligned StOp, only two scan chains are needed since their age relative to only two categories must be determined:

1. any Stage2 LdOp or Stage1 LdOp performing the second half of a misaligned LdOp.
2. any pre-Stage2 LdOps.

Relative age determination for Stage2 LdOps is possible, at the cost of a third scan chain, but proves to be of insignificant performance benefit. This is because StOps: (a) have less opportunity to try to and execute ahead of older LdOps and (b) often do not have following dependent LdOps that would benefit from earlier StOp execution.

As in the Op issue selection scans, each scan handles one of the cases identified above. Then, based on the value of the carry-in signals to the Op entry holding the StOp in question, StOp ordering control logic can determine whether to examine the Load Unit pipeline Stage2 address comparator and then whether to hold up the StOp.

The scan chains are simple propagate-kill chains, scanning from the oldest scheduler Op entry to the youngest. The following pseudo-RTL description describes each of the five scan chains in terms of carry lookahead equations. The bit-level P and K terms are based only on the State and Type fields of an Op entry (see the section titled “Dynamic Field State[3:0]” beginning on page 205 and the section titled “Dynamic Field Exec1” beginning on page 209, respectively). For the three LdOp scan chains, the ST Type field value (101) is used instead of the SU Type field value (10X) as this distinguishes the StOps which

[29] See the section titled “StOp Misaligned Accesses” beginning on page 173.

actually reference memory from LEA Ops, which only generate logical addresses. LUst2, LUst1, LUst0, SUld2, and SUld1 denote the five scan chains.

Pseudo-RTL Description

Ld-St Ordering Determination Logic

```
Bit-level or Op entry equations:

LUst2:~P = K = ST & ~S3 & (S2 | S1 & SU2_FirstAddrV)
LUst1:~P = K = ST & ~S2
LUst0:~P = K = ST & ~S1
SUld2:~P = K = LU & ~S3 & (S2 | S1 & LU2_FirstAddrV)
SUld1:~P = K = LU & ~S2


Group lookahead equations (based on four-bit groups):

Pgrp = P0 & P1 & P2 & P3

CIn0 = CIn// note: Op 0 is oldest Op within a quad
CIn1 = CIn & P0
CIn2 = CIn & P0 & P1
CIn3 = CIn & P0 & P1 & P2

Lookahead among Quads:

CinGrp5 = 1// note: Quad 5 is oldest quad
CinGrp4 = Pgrp5
CinGrp3 = Pgrp5 & Pgrp4
CinGrp2 = Pgrp5 & Pgrp4 & Pgrp3
CinGrp1 = Pgrp5 & Pgrp4 & Pgrp3 & Pgrp2
CinGrp0 = Pgrp5 & Pgrp4 & Pgrp3 & Pgrp2 & Pgrp1

During the second phase of pipe stage 2 for a LdStOp, the two/three Cin's to
the Op entry holding the LdStOp are combined with a 24:1 mux as follows:

LUAges[2:0] = 3'b000
SUAges[1:0] = 2'b00

for (all Ops)
 LUAges[2:0] |=
    (LU & ~S3 & (S2 | S1 & LU2_FirstAddrV)) ?
      {~LUst2chain.CIN,~LUst1chain.CIN,~LUst0chain.CIN} : 3'b0
  SUAges[1:0] |=
    (SU & ~S3 & (S2 | S1 & SU2_FirstAddrV)) ?
      {~SUld2chain.CIN,~SUld1chain.CIN} : 2'b0
```

Scheduler Op Entry Fields Read Out During Operand Transfer

During the Operand Information Broadcast and the Operand Transfer Phases of fetching operand values, a variety of information is read out from the associated Ops for use by external control logic. For most operands this occurs during the Op Issue Stage and pipeline Stage0 of the processing pipelines. For the store data operand of StOps this occurs during SU pipeline Stage1 and Stage2.

During the operand information broadcast phase, information about the Op whose operands are being fetched is read out onto the appropriate OpInfo bus. In parallel, the SrcReg and SrcBM fields of the Op's scheduler entry are read out onto the two associated OprndInfo buses. In the case of the store data operand for a StOp, there is not an associated OpInfo bus transaction since this information is retained from when the StOp was issued. The OprndInfo information is used during the next couple of phases. The OpInfo data is simply passed down the pipeline to the actual execution units; in the case of RUX and RUY, the two source BM[0] bits from the OprndInfo buses are also passed down the pipeline to the execution units.

During the operand transfer phase, information about the status of each Op that is the source of an operand value is read out onto the OprndStat bus associated with each Oprnd bus. This information is only used during operand transfer phase. The following pseudo-RTL description summarizes all of the information that is read out of the scheduler, at various times, for external use. In these equations, XX = {LU, SU, RUX, RUY} and Y = {1, 2}.

Pseudo-RTL Description

Scheduler Information for External Use

```
During Operand Information Broadcast phase:
     OprndInfo_XXsrcY[7:0]
     OprndInfo_SusrcSt[7:0]
     SrcYReg[4:0]
     SrcYBM[2:0]
     OpInfo_XX[12:0]

During Operand Transfer phase:
     OprndStat_XXsrcY[9:0]
     OprndStat_SUsrcSt[9:0]
     State[3:0]
     DestBM[2:0]
     Type[2:1]
     Exec1
```

Global Control Logic

During the descriptions of the logic, storage elements, and buses comprising the core of the scheduler, there has been reference to a modest but critically important amount of peripheral control logic that coordinates the overall operation of the scheduler and the "feeding" of Ops to the four execution pipelines. The section describes each piece of this peripheral logic in its order of significance within the four phases of the first two pipeline stages.

During the issue selection phase the only external question to resolve is whether any Op had been selected for issue to the various execution unit processing pipelines. For each issue selection that did not find an eligible Op, the corresponding OprndInfo and OpInfo buses will not be driven by a scheduler entry. In such cases, the values on these buses and the operation of the scheduler during the following three phases for this execution unit processing pipeline in question is a "don't care." The only requirement is that the scheduler must know if it should treat pipeline Stage0 as still being empty or not. This is accomplished by using OpValid bits, one for each pipeline stage.

OpValid bits:
OpV_LU_0
OpV_SU_0
OpV_RUX_0
OpV_RUY_0

The Stage0 OpValid bits are called OpV_LU_0, OpV_SU_0, OpV_RUX_0, and OpV_RUY_0. Thus the OpValid bit passed into pipeline Stage0 must be zero for this pipeline stage is to be treated as still being empty. The OpValid bits for each processing pipeline are generated from the final carry-out (C_{out}) of the issue selection scan chains out of the youngest Op's scheduler entry. The OpValid bits are loaded into pipeline registers controlled by the XXAdv0 global signals. In addition, during abort cycles these registers are unconditionally cleared. The following pseudo-RTL description summarizes these equations:

Pseudo-RTL Description

Op_XXX_Iss Signals

```
OpV_LU_Iss  = ~LUchain.COUT
OpV_SU_Iss  = ~SUchain.COUT
OpV_RUX_Iss = ~RUXchain.COUT
OpV_RUY_Iss = ~RUYchain.COUT
```

During the Operand Information Broadcast Phase there is no significant peripheral logic other than the pipeline registers which latch the OprndInfo and OpInfo values read out of the scheduler for external use. During the operand selection phase two external activities take place:

1. the SrcYReg fields of the latched OprndInfo values (i.e., the source register numbers read out during the preceding phase) are used to access the architectural register file. This is done blindly and in

parallel with operation of the operand selection scan chains within the scheduler.

2. the determination is made for each operand bus of whether the register file or the scheduler is to drive the bus with an operand value during the next phase. Each scheduler Op entry directly determines for itself whether it should drive the bus or not, so the only issue is determining whether the register file should be enabled to drive the bus. This decision is simply based on whether any Op entry was selected during this phase or not. If no Op entry was selected, as indicated by the final C_{out} of the associated operand selection scan chains, then the register file is enabled. This is done independently for each operand bus and the equations describing this are summarized in the operand selection section.

DESIGN NOTE

Architectural Register File Ports

Since there are up to nine source operands[a] to be fetched each cycle, there are nine corresponding read ports to the architectural register file, each port being associated with a source to one of the operand buses. The register fields presented to these ports are SUsrcSt and XXsrcY, where XX = {LU, SU, RUX, RUY} and Y = {1, 2}.

[a] Eight Stage0 operands plus the store data operand.

During the operand transfer phase there are a number of external control functions that occur:

1. RegOp bumping.
2. control of all the execution unit operand input multiplexers.
3. validity determination for each operand value being fetched.
4. generation of the HoldXX0 signals.[30]

Each of these situations will now be discussed

RegOp Bumping

As described in the section titled “RegOp Bumping” beginning on page 246, the implementation of RegOp bumping is split between logic within each scheduler Op entry and peripheral logic which generates the global

[30] The HoldXX0 signal factor into the generation of the XXAdv0 global pipeline register control signals.

bump signals BumpRUX and BumpRUY and forces assertion of the RUXAdv0 and RUYAdv0 signals. The generation of the BumpRUX and BumpRUY signals are based on the OprndStat values that are read out of the scheduler, during the operand transfer phase, for each of the register unit source operands—i.e., the four values OprndStat_RUXsrcY, where X, Y = {1, 2}. In particular, the State and Type fields for each operand source are examined to determine whether the sourcing Op is at least two cycles away from being able to (possibly) provide a valid operand value. If this is the case for either sourcing Op, then the dependent RegOp is bumped out of pipeline Stage0.

DEFINITION

Sourcing Op

A Sourcing Op is an Op that has been selected to be the source of an operand value.

The following pseudo-RTL description summarizes the BumpRUX and BumpRUY equations and includes an additional term in each equation (the RUXTimeout and RUYTimeout signals) to handle what could otherwise be deadlock situations:

PSEUDO-RTL DESCRIPTION

RegOp Bumping Logic

```
Inhibit if RUX-only stage 0 RegOp and valid RUX-only issue stage RegOp:

InhBumpRUX = OpInfo_RUX(RegOp).R1 & OpV_RUX_Iss & OpInfo_RUX_0(RegOp).R1

// "~S0 | ~S1 & LU | timeout"
BumpRUX = ~InhBumpRUX &
          (~OprndStat_RUXsrc1.State[0] |
           (~OprndStat_RUXsrc1.State[1] & ~OprndStat_RUXsrc1.Type[1]) |
           ~OprndStat_RUXsrc2.State[0] |
           (~OprndStat_RUXsrc2.State[1] & ~OprndStat_RUXsrc2.Type[1]) |
           RUXTimeout)

// "~S0 | ~S1 & LU | timeout"
BumpRUY =   ~OprndStat_RUYsrc1.State[0] |
   (~OprndStat_RUYsrc1.State[1] & ~OprndStat_RUYsrc1.Type[1]) |
    ~OprndStat_RUYsrc2.State[0] |
   (~OprndStat_RUYsrc2.State[1] & ~OprndStat_RUYsrc2.Type[1]) |
    RUYTimeout
```

The RUXTimeout and RUYTimeout terms are generated by 3-bit counters associated with Stage0 of the RUX and RUY pipelines. Taking RUX as an example, whenever RUX Stage0 is loaded, irrespective of whether with a valid or invalid Op, the associated counter is reset to a start value; during all other cycles the counter is decremented. If the counter reaches 000, then RUXTimeout is asserted.

The BumpRUX and BumpRUY signals force the reload of Stage0 of the RUX and RUY pipelines, and reset the State bits to unissued within the scheduler Op entries corresponding to the RegOps being bumped. They also deassert the Stage0 OpValid signals and thus prevent a Stage0 RegOp, while being bumped, from also advancing into pipeline Stage1. For example, RUXTimeout immediately forces OpV_RUX_0 = 0. Recall that if the Stage0 OpValid bit appears to be zero, the pipeline stage will be treated as being empty. OpV_RUX_0 = 0 then causes assertion of the RUXAdv0 pipeline control signal and thus reloading of RUX Stage0.

Control of All Execution Unit Operand Input Multiplexers

The second peripheral function occurring during the operand transfer phase is generation of the controls for each of the source operand input multiplexers of each of the execution units. There are nine such multiplexers, one for each of the eight Stage0 operands plus one for a store data operand. As described in an earlier section, there is a 5:1 multiplexer associated with each operand fetch that selects a value from either the corresponding operand bus or one of the four execution result buses, to load into the execution unit's operand register.

The control for each of these multiplexers is based on the OprndStat values that are read out of the scheduler, during operand transfer phase, for each of the operands being fetched, i.e., OprndStat_XXsrcY and OprndStat_SUsrcSt, where XX = {LU, SU, RUX, RUY} and Y = {1, 2}. In particular, the State and Type fields for each operand source are examined to determine whether the sourcing Op has already completed execution or, if not, then which unit it is being executed by. The case of a RegOp sourcing a Src2 immediate value to itself is also appropriately handled. The following pseudo-RTL description summarizes the five input select equations for each operand multiplexer:

.

Pseudo-RTL Description

Operand Multiplexer Logic

```
// RUXsrc2 OprndStat values
RUXsrc2Imm = (Type[1:0]=2'b11) & ~S1 & S0 &  Exec1

// RUYsrc2 OprndStat values
RUYsrc2Imm = (Type[1:0]=2'b11) & ~S1 & S0 & ~Exec1

// “S3 | S2 & ~LU”
SelOprndBus_XXsrcY = State[3] | State[2] & Type[1]

// above is for all SelOprndBus signals except the two below
SelOprndBus_RUXsrc2 = State[3] | State[2] & Type[1] | RUXsrc2Imm
SelOprndBus_RUYsrc2 = State[3] | State[2] & Type[1] | RUYsrc2Imm

SelLUDestRes_XXsrcY  = ~SelOprndBus_XXsrcY & ~Type[1]
SelSUDestRes_XXsrcY  = ~SelOprndBus_XXsrcY &  Type[1] & ~Type[0]
SelRUXDestRes_XXsrcY = ~SelOprndBus_XXsrcY &  Type[1] &  Type[0] &  Exec1
SelRUYDestRes_XXsrcY = ~SelOprndBus_XXsrcY &  Type[1] &  Type[0] & ~Exec1
```

Validity Determination for Each Operand Value Being Transferred

The third peripheral function occurring during the operand transfer phase is determination of the validity of each of the nine operand values being presented to execution unit source operand registers. A signal is generated for each source operand that indicates whether its current value is valid or not. As with the control of the associated execution unit input multiplexers, the signal is based on the State and Type fields of the OprndStat values that are read out of the scheduler. The sourcing Op must either have completed execution or currently be completing execution. In addition, the DestBM field of the OprndStat value is compared with the Src1BM or Src2BM field of the latched OprndInfo value for the operand being fetched. The sourcing Op's byte marks must be a superset of the required byte marks. The case of a RegOp sourcing a Src2 immediate value to itself is also appropriately handled, but in the HoldXX0 equations described below instead of being factored into the appropriate equations for the OprndInvld signals. The following pseudo-RTL descriptions summarize the OprndInvld equations:

PSEUDO-RTL DESCRIPTION

OprndInvld Signals

```
//~S1* | LU & (~S2 | ~S3 & ~LUAdv2)
OprndInvld_XXsrcY= ~State[1] |
                   ~Type[1] & (~State[2] | ~State[3] & ~CHP_LUAdv2) |
                   SrcYBM[2] & ~DestBM[2] |
                   SrcYBM[1] & ~DestBM[1] |
                   SrcYBM[0] & ~DestBM[0]
```

Generation of the HoldXX0 Signals

The fourth and last peripheral function occurring during the operand transfer phase is generation of the HoldXX0 pipeline control signals. The following summarizes the equations for generating these signals:

PSEUDO-RTL DESCRIPTION

HoldXX0 Pipeline Control Signals

```
SC_HoldLU0  = OprndInvld_LUsrc1 | OprndInvld_LUsrc2
SC_HoldSU0  = OprndInvld_SUsrc1 | OprndInvld_SUsrc2
SC_HoldRUX0 = OprndInvld_RUXsrc1 | OprndInvld_RUXsrc2 & ~RUXsrc2Imm |
              StatusInvld_RUX | NonAbSync
SC_HoldRUY0 = OprndInvld_RUYsrc1 | OprndInvld_RUYsrc2 & ~RUYsrc2Imm
SC_HoldSU2  = OprndInvld_SUsrcSt & ~SU2_FirstAddrV
```

STATUS FLAG HANDLING LOGIC, STATUS FLAG DEPENDENT REGOP LOGIC, BRANCH RESOLUTION LOGIC, AND NONABORTABLE REGOP LOGIC

The handling and usage of status flags, both architectural status flags and K6 microarchitectural flags, involves three areas of functionality:

1. fetching of status flag operand values for cc-dependent RegOps.
2. fetching of status flag values for and the resolution of BRCOND Ops.
3. synchronization of the execution of nonabortable RegOps.

Unlike the handling of register operands and LdOp-StOp ordering constraints, the logic for supporting these functions is not spread across all scheduler Op entries. Status flag handling for related Ops can only occur while they are within certain rows of the scheduler. In the case of cc-dependent RegOps, they must be in Row 3 during the RUX pipeline Stage0 cycle (i.e., the cycle in which status operand fetching occurs). In the case of BRCOND Ops and nonabortable RegOps, they must be in Row 4 during their resolution or RUX Stage0 cycle, respectively.

Design Note

Simplifications and Reductions in Logic

Condition code dependent and nonabortable RegOps are held up in RUX Stage0 if they have not yet shifted down to scheduler Row 3 or 4, respectively. Conversely, such Ops are held up at these positions by inhibiting OpQuad shifting until they are successfully able to advance into RUX Stage1. These restrictions enable all of the associated logic to be simpler and much smaller. For example, the fetching of appropriate status flag values for condition code dependent RegOps and BRCOND Ops only occurs across the bottom three scheduler rows and can be performed independently for each of the four groups of status flags. One set of this status fetching logic can be shared or utilized for both condition code dependent RegOp status operand fetching and BRCOND Op resolution.

In addition, the direct bypassing of status flag values directly from either RegOp execution unit to a condition code dependent RegOp entering the RUX execution unit, is not supported. The result is a minimum one-cycle latency between the execution of a RegOp that modifies status flags[a] and the execution of a following cc-dependent RegOp. The overall performance impact of this is found to be minimal. In cases where an OpQuad Sequence is being executed, the impact might very well be eliminated through appropriate Op scheduling.

[a] Such RegOps are termed ".cc" RegOps.

To further aid the simplification and reduction of logic, a number of restrictions are placed on where condition code dependent RegOps, BRCOND Ops, and nonabortable RegOps can occur relative to each other within OpQuads. Many of the relevant Ops can only occur in OpQuad Sequences. The restrictions generally translate into OpQuad Sequence coding rules and in some cases also constrain the decoding of multiple x86 instructions in one cycle. In particular, the restrictions are as follows:

1. no ".cc" RegOps after a BRCOND Op within an OpQuad.
2. no cc-dependent RegOps after a ".cc" RegOp within an OpQuad.
3. no nonabortable RegOps in the same OpQuad as a BRCOND Op.

4. only one cc-dependent RegOp within an OpQuad.
5. only one BRCOND Op within an OpQuad.

The following describes each of the pieces of logic: for status value fetching, status forwarding to condition code dependent RegOps, BRCOND resolution, and nonabortable RegOp synchronization

The Fetching of Status Flag Operand Values for CC-Dependent RegOps and BRCOND Ops

During each cycle, the effective set of status flag values at the boundary between scheduler Rows 3 and 4 is computed by examining all the RegOps in OpQuads 4 and 5. Since it is possible for each RegOp to modify only a subset of the flags, this process is performed independently for each of four groups of status flags, corresponding to the four StatMod bits within each scheduler Op entry (see the section titled "Dynamic Field StatMod[3:0]" beginning on page 211). The result, within each group, is a set of flag values and state information from the youngest RegOp with its StatMod bit set for that set of flag values. The validity of the flag values is directly implied by the associated state information.

The end result or output of the status flag fetch logic is eight flag values and four associated valid bits. These are passed to the logic handling condition code dependent RegOps and to the logic handling BRCOND Op resolution, where the flag values are evaluated and the valid bits are examined to determine whether the required flag values are, in fact, valid. Based on the latter information, appropriate pipeline and scheduler shift control signals are generated. The correspondence between the StatMod[3:0] bits and the status flags was given in Table 3.9 on page 211.

NOTATION (REPEATED FOR CONVENIENCE)

OpQuadY and OpX

We will, from time to time, use the following notation to refer to scheduler OpQuads: OpQuadY, where Y = 0 to 5. For example, OpQuad1 identifies the OpQuad in Row 1 of the scheduler, OpQuad2 the OpQuad in Row 2, etc. Additionally, we will use the following notation to refer to scheduler Op entries: OpX, where X = 0 to 23. For example, X = 0 identifies the youngest Op in the scheduler and X = 23 identifies the oldest Op in the scheduler. Thus OpQuad4, for example, contains Op16, Op17, Op18, and Op19.

A process somewhat similar to that for fetching register operand values is used within each status flag group to obtain the appropriate flag values, i.e., the most recent new values relative to Op15 in the scheduler. A propagate-kill-style scan from Op16 to Op23 locates the first Op with its StatMod bit for this flag group set, and that Op entry's Completed State bit (i.e., S3) and the appropriate set of flag values are read out; the valid bit for this group is simply the State bit. In the case that no such Op is found, the desired flag values are read from the architectural status flags register along with S3 = 1. The following pseudo-RTL descriptions give the equations for the status flag fetch logic for each flag:

PSEUDO-RTL DESCRIPTION

Status Flag Fetching Logic

```
for (j = 16 to 23) {
  Op[j]:StatInfo_3[1:0] = {Op[j]:StatVal[7],  Op[j]:S3} //OF
  Op[j]:StatInfo_2[4:0] = {Op[j]:StatVal[6:3],Op[j]:S3} //SF,ZF,AF,PF
  Op[j]:StatInfo_1[1:0] = {Op[j]:StatVal[2],  Op[j]:S3} //CF
  Op[j]:StatInfo_0[2:0] = {Op[j]:StatVal[1:0],Op[j]:S3} //EZF,ECF
}

FlgStatInfo_3[1:0] = {StatFlags[7],  1'b1} //OF
FlgStatInfo_2[4:0] = {StatFlags[6:3],1'b1} //SF,ZF,AF,PF
FlgStatInfo_1[1:0] = {StatFlags[2],  1'b1} //CF
FlgStatInfo_0[2:0] = {StatFlags[1:0],1'b1} //EZF,ECF

for (i = 0 to 3) {
  // i indexes flag group corresponding to StatMod[i]
  StatSel16_[i] =  Op16:StatMod[i]
  StatSel17_[i] = ~Op16:StatMod[i] & Op17:StatMod[i]
  StatSel18_[i] =
    ~Op16:StatMod[i] & ~Op17:StatMod[i] & Op18:StatMod[i]
  StatSel19_[i] =
    ~Op16:StatMod[i] ... ~Op18:StatMod[i] & Op19:StatMod[i]
  StatSel20_[i] =
    ~Op16:StatMod[i] ... ~Op19:StatMod[i] & Op20:StatMod[i]
  StatSel21_[i] =
    ~Op16:StatMod[i] ... ~Op20:StatMod[i] & Op21:StatMod[i]
  StatSel22_[i] =
    ~Op16:StatMod[i] ... ~Op21:StatMod[i] & Op22:StatMod[i]
  StatSel23_[i] =
    ~Op16:StatMod[i] ... ~Op22:StatMod[i] & Op23:StatMod[i]
  StatSelFlg_[i]=
    ~Op16:StatMod[i] ... ~Op22:StatMod[i] & ~Op23:StatMod[i]
}
```

continued on next page ...

Pseudo-RTL Description (cont.)

Status Flag Fetching Logic

```
StatInfo_3[1:0] =  {2{StatSel16_3}} & Op16:StatInfo_3[1:0] |
                   {2{StatSel17_3}} & Op17:StatInfo_3[1:0] |
                   {2{StatSel18_3}} & Op18:StatInfo_3[1:0] |
                   {2{StatSel19_3}} & Op19:StatInfo_3[1:0] |
                   {2{StatSel20_3}} & Op20:StatInfo_3[1:0] |
                   {2{StatSel21_3}} & Op21:StatInfo_3[1:0] |
                   {2{StatSel22_3}} & Op22:StatInfo_3[1:0] |
                   {2{StatSel23_3}} & Op23:StatInfo_3[1:0] |
                   {2{StatSelFlg_3}} & FlgStatInfo_3[1:0]

StatInfo_2[4:0] =  {5{StatSel16_2}} & Op16:StatInfo_2[4:0] |
                   {5{StatSel17_2}} & Op17:StatInfo_2[4:0] |
                   {5{StatSel18_2}} & Op18:StatInfo_2[4:0] |
                   {5{StatSel19_2}} & Op19:StatInfo_2[4:0] |
                   {5{StatSel20_2}} & Op20:StatInfo_2[4:0] |
                   {5{StatSel21_2}} & Op21:StatInfo_2[4:0] |
                   {5{StatSel22_2}} & Op22:StatInfo_2[4:0] |
                   {5{StatSel23_2}} & Op23:StatInfo_2[4:0] |
                   {5{StatSelFlg_2}} & FlgStatInfo_2[4:0]

StatInfo_1[1:0] =  {2{StatSel16_1}} & Op16:StatInfo_1[1:0] |
                   {2{StatSel17_1}} & Op17:StatInfo_1[1:0] |
                   {2{StatSel18_1}} & Op18:StatInfo_1[1:0] |
                   {2{StatSel19_1}} & Op19:StatInfo_1[1:0] |
                   {2{StatSel20_1}} & Op20:StatInfo_1[1:0] |
                   {2{StatSel21_1}} & Op21:StatInfo_1[1:0] |
                   {2{StatSel22_1}} & Op22:StatInfo_1[1:0] |
                   {2{StatSel23_1}} & Op23:StatInfo_1[1:0] |
                   {2{StatSelFlg_1}} & FlgStatInfo_1[1:0]

StatInfo_0[2:0] =  {3{StatSel16_0}} & Op16:StatInfo_0[2:0] |
                   {3{StatSel17_0}} & Op17:StatInfo_0[2:0] |
                   {3{StatSel18_0}} & Op18:StatInfo_0[2:0] |
                   {3{StatSel19_0}} & Op19:StatInfo_0[2:0] |
                   {3{StatSel20_0}} & Op20:StatInfo_0[2:0] |
                   {3{StatSel21_0}} & Op21:StatInfo_0[2:0] |
                   {3{StatSel22_0}} & Op22:StatInfo_0[2:0] |
                   {3{StatSel23_0}} & Op23:StatInfo_0[2:0] |
                   {3{StatSelFlg_0}} & FlgStatInfo_0[2:0]
```

continued on next page ...

Pseudo-RTL Description (cont.)

Status Flag Fetching Logic

```
// OF; SF,ZF,AF,PF; CF; EZF,ECF
Status[7:0] =
  {StatInfo_3[1],StatInfo_2[4:1],StatInfo_1[1],StatInfo_0[2:1]}

StatusV[3:0] =
  {StatInfo_3[0],StatInfo_2[0],StatInfo_1[0],StatInfo_0[0]}
```

CC-Dependent RegOp Synchronization

During each cycle, the four Ops within scheduler Row 3 are examined for whether any of them is a condition code dependent RegOp. If one is found, then the specific type of the RegOp is decoded to determine which groups of status flags are needed, and the Status valid bits are checked to determine whether all of those groups are, in fact, valid. Concurrently, bits Status[7:0] are blindly passed to the RUX execution unit.

If all of the required flag groups are currently valid, then the RegOp is allowed to advance into RUX pipeline Stage1, at least insofar as the status operand fetch is concerned. If the RegOp does not immediately advance into Stage1, though, then shifting of scheduler OpQuad3—and thus Opquad4-OpQuad5—are inhibited. If any of the required flag groups are not currently valid, then the RegOp is held up from advancing into RUX Stage1 and shifting of scheduler OpQuad3-OpQuad5 is inhibited.

If there is no unexecuted cc-dependent RegOp in scheduler OpQuad3, but there is a cc-dependent RegOp in RUX pipeline Stage0, then the RegOp is unconditionally held up in Stage0 until it also arrives in OpQuad3. If there is a cc-dependent RegOp in OpQuad3 that has not yet executed, but there is no cc-dependent RegOp in RUX Stage0 or there is an unexecuted cc-dependent RegOp in scheduler OpQuad4, then shifting of OpQuad3-OpQuad5 is inhibited.

There is an additional input called RUX_NoStatMod from the RUX unit pipeline Stage 1. RUX_NoStatMod indicates that the Op being executed there does not modify status flags despite it being marked as modifying status flags. This is necessary to handle certain architectural situations where a RegOp does not modify status flags for certain zero operand values. A cycle-delayed version, called NoStatMod, is used in control logic. The following pseudo-RTL description gives the equations for synchronizing or coordinating the execution of cc-dependent RegOps:

Pseudo-RTL Description

CC-Dependent RegOp Synchronization Logic

```
CCDepInRUX_0 = (OpInfo_RUX_0(RegOp).Type[3:2] = 2'b01) & OpV_RUX_0

UnexecCCDepInQ3 = OP12:(RU & OpInfo(RegOp).Type[3:2]=2'b01 & ~S1) |
                  OP13:(RU & OpInfo(RegOp).Type[3:2]=2'b01 & ~S1) |
                  OP14:(RU & OpInfo(RegOp).Type[3:2]=2'b01 & ~S1) |
                  OP15:(RU & OpInfo(RegOp).Type[3:2]=2'b01 & ~S1)

if (~OpInfo_RUX_0(RegOp).Type[5])
  StatV = StatusV[1]  //need CF for ADC,SBB,RLC,RRC Ops

elseif (OpInfo_RUX_0(RegOp).Type[1:0] = 2'b10)
  StatV = StatusV[0]  //need EZF,ECF for MOVcc Op

else                  //need OF,...,CF for MOVcc,RDFLG,DAA,DAS Ops
  StatV = StatusV[3] & StatusV[2] & StatusV[1]

// keep track of when an unexecuted cc-dep RegOp
// is in Sched Op quad 3:

StrtExecCCDep = CCDepInRUX_0 & SC_AdvRUX0 & ~BumpRUX

// keep track of when an unexecuted cc-dep RegOp
// is in Sched Op quad 4
if (LdEntry4 | StrtExecCCDep | SC_EAbort)  // enabled flip-flop
  UnexecCCDepInQ4 = LdEntry4 & UnexecCCDepInQ3 &
                    ~StrtExecCCDep & ~SC_EAbort
// hold copy of status flag values at input to
// RUX execution unit:
SC_HoldStatus = UnexecCCDepInQ4

// hold RegOp execution if ...:
StatusInvld_RUX = (CCDepInRUX_0 & ~UnexecCCDepInQ4) &
  ~(UnexecCCDepInQ3 & StatV & ~NoStatMod)

// hold Op quad from shifting out of Sched quad 3 if ...:
HoldOpQ3 = UnexecCCDepInQ3 & ~(CCDepInRUX_0 & StatV & ~NoStatMod) |
           UnexecCCDepInQ4
```

BRCOND Op Resolution Logic

In the "Branch Resolution Logic" section of Chapter 2, we explained that a BRCOND Op must be resolved as to whether the associated conditional branch instruction was correctly predicted or not while the BRCOND Op is outstanding *and* before it reaches the bottom row of the scheduler's buffer. By way of review, we will summarize that discussion before presenting the pseudo-RTL description which gives the equations for the BRCOND Op resolution logic. We also recommend that you review the placement of the Branch Resolution Unit (a.k.a. the branch resolution logic) in Figure 2.2 on page 69 and in Figure 2.9 on page 130.

BRCOND Op resolution is done for each branch operation, in order, at a rate of up to one per cycle. The appropriate set of status flag values is used to determine if the condition code specified within the BRCOND Op is TRUE or FALSE when the status flag handling logic obtains the appropriate status for the next unresolved BRCOND Op. If valid values for the required status flags are not yet all available, then resolution of the BRCOND Op is held up.

If the branch condition is FALSE, the BRCOND Op was incorrectly predicted and an appropriate restart signal is immediately asserted to restart the upper portion of the processor at the correct next program address (i.e., the instruction fetch and decode portion—see Figure 2.4 on page 87). The correct address is either the branch target address or next sequential address, whichever was not predicted.

If the branch was correctly predicted, then nothing happens other than BRCOND Op resolution processing advances on to the next BRCOND Op. The branch resolution logic is concerned with resolving these issues.

With this review as background, we will now give some additional, more detailed comments, to aid in understanding the pseudo-RTL description of the resolution process.

During each cycle, the four Ops within scheduler OpQuad4 are examined for whether any of them (at most one) is a BRCOND Op. If one is found, then the Condition Code field of that Op entry is decoded to select one of thirty-two condition values and associated valid bits. The value and validity of the selected condition are then used to inhibit scheduler shifting of OpQuad4-OpQuad5 or to assert pipeline restart signals when appropriate.

In the case that a BRCOND Op is found to be mispredicted, a pipeline restart is required. The appropriate restart signal is asserted based on whether the BRCOND Op has been produced by the decoders or is from an OpQuad Sequence. If the BRCOND Op is from an OpQuad Sequence, the signal also depends on whether it is from an internal (OpQuad ROM-based) or external (external memory-based) sequence. In addition to generating the restart signal, an appropriate x86 instruction address or OpQuad Sequence vector address must be generated.

For the benefit of the logic handling synchronization between non-abortable RegOps and preceding BRCOND Ops, a record is also maintained of the occurrence of a mispredicted BRCOND Op until an abort cycle occurs via the SC_MisPred flip-flop. Further, the existence of an outstanding mispredicted BRCOND Op is used to hold up the loading of new OpQuads into the scheduler from the "restarted" decoders until the abort cycle occurs.

In the case that a BRCOND Op was correctly predicted, the only action taken is to set the BRCOND Op's S3 State bit. The following pseudo-RTL equations describe all of this logic. Reference is made below to the DTF and SSTF signals used to indicate pending data breakpoint and single-step traps, respectively. There is also a signal called MDD ("multiple decode disable") which can be used for silicon debugging to prevent more than one instruction from being decoded into each OpQuad. The following pseudo-RTL description gives the equations for the BRCOND Op resolution logic:

Pseudo-RTL Description

BRCOND Op Resolution Logic

```
BRCOND16 = OP16:(Type=SpecOp & OpInfo(SpecOp).Type=BRCOND & ~S3)
BRCOND17 = OP17:(Type=SpecOp & OpInfo(SpecOp).Type=BRCOND & ~S3)
BRCOND18 = OP18:(Type=SpecOp & OpInfo(SpecOp).Type=BRCOND & ~S3)
BRCOND19 = OP19:(Type=SpecOp & OpInfo(SpecOp).Type=BRCOND & ~S3)

BRCONDInQ4 = (BRCOND16 | BRCOND17 | BRCOND18 | BRCOND19) & OPQ4:OpQV

CondCode[4:0] = {5{BRCOND16}} & Op16:OpInfo(SpecOp).CC[4:0] |
                {5{BRCOND17}} & Op17:OpInfo(SpecOp).CC[4:0] |
                {5{BRCOND18}} & Op18:OpInfo(SpecOp).CC[4:0] |
                {5{BRCOND19}} & Op19:OpInfo(SpecOp).CC[4:0]

CondV = switch (CondCode[4:1])
          case 0000:  1'b1
          case 0001:  StatusV[0]
          case 0010:  StatusV[0]
          case 0011:  StatusV[0] & StatusV[2]
          case 0100:  StatusV[0]
          case 0101:  StatusV[0]
          case 0110:  StatusV[0]
          case 0111:  StatusV[0] & StatusV[2]
          case 1000:  StatusV[3]
          case 1001:  StatusV[1]
          case 1010:  StatusV[2]
          case 1011:  StatusV[2] & StatusV[1]
          case 1100:  StatusV[2]
          case 1101:  StatusV[2]
          case 1110:  StatusV[3] & StatusV[2]
          case 1111:  StatusV[3] & StatusV[2]

// any active hardware interrupt requests?:
IP = SI_NMIP | SI_INTRP
```

continued on next page ...

Pseudo-RTL Description

BRCOND Op Resolution Logic

```
CondVal = switch (CondCode[4:1])
            case 0000: CondCode[0] ^ 1'b1
            case 0001: CondCode[0] ^ Status[0]
            case 0010: CondCode[0] ^ Status[1]
            case 0011: Status[1] | (CondCode[0] ^ ~Status[5])
            case 0100: CondCode[0] ^ (~Status[1] & ~IP & ~(DTF|SSTF|MDD))
            case 0101: CondCode[0] ^ (~Status[1] & ~IP & ~(DTF|SSTF|MDD))
            case 0110: CondCode[0] ^ (~Status[0] & ~IP & ~(DTF|SSTF|MDD))
            case 0111: ~Status[1] & ~IP & ~(DTF|SSTF|MDD) &
                       (CondCode[0] ^ Status[5])
            case 1000: CondCode[0] ^ Status[7]
            case 1001: CondCode[0] ^ Status[2]
            case 1010: CondCode[0] ^ Status[5]
            case 1011: CondCode[0] ^ (Status[5] | Status[2])
            case 1100: CondCode[0] ^ Status[6]
            case 1101: CondCode[0] ^ Status[3]
            case 1110: CondCode[0] ^  (Status[7] ^ Status[6])
            case 1111: CondCode[0] ^ ((Status[7] ^ Status[6]) | Status[5])

// the definitions of CondCode[4:1] is as follows
// (bit 0 flips the sense):

True    4'b0000
ECF     4'b0001
EZF     4'b0010
SZnZF   4'b0011
MSTRZ   4'b0100
STRZ    4'b0101
MSTRC   4'b0110
STRZnZF 4'b0111
OF      4'b1000
CF      4'b1001
ZF      4'b1010
CvZF    4'b1011
SF      4'b1100
PF      4'b1101
SxOF    4'b1110
SxOvZF  4'b1111
```

continued on next page...

Pseudo-RTL Description (cont.)

BRCOND Op Resolution Logic

```
// hold Op quad from shifting out of Sched quad 4 if ...:
HoldOpQ4A = BRCONDInQ4 & ~CondV

SC_Resolve = BRCONDInQ4 & CondV & ~SC_MisPred & ~NoStatMod & ~OPQ4:Emcode

// remember resolution of a BRCOND Op in quad 4:
Resolved = ~LdEntry4 & (SC_Resolve | Resolved)// simple flip-flop

// terminate REP MOVS OpQuad Sequence loop if almost done:
// use CS “D” bit supplied by RUX to aid termination in 16-bit case
TermMovs = BRCONDInQ4 & CondV & ~NoStatMod & ~SC_MisPred &
           // CondCode=MSTRC ... | CondCode=MSTRZ ...
           ((CondCode[4:1] = 4'b0110) & (OP19:DestVal[15:0] = 16'h5) &
            ((OP19:DestVal[31:16] = 16'h0) | RUX_D) |
            (CondCode[4:1] = 'b0100) (OP23:DestVal[15:0] = 16'h6) &
            ((OP23:DestVal[31:16] = 16'b0) | RUX_D))

TermedMOVS = ~LdEntry4 & (TermMOVS || TermedMOVS)// simple flip-flop

SC_TermMOVS = TermMOVS | TermedMOVS

// get OpQuad Sequence or instruction vector address for
// handling mispredicted branch
BrVecAddr[31:0] = {32{BRCOND16}} & Op16:DestVal[31:0] |
  {32{BRCOND17}} & Op17:DestVal[31:0] |
  {32{BRCOND18}} & Op18:DestVal[31:0] |
  {32{BRCOND19}} & Op19:DestVal[31:0]

// supply old RAS TOS ptr to decoders for restoring
// if BRCOND Op mispredicted:
SC_OldRASPtr[2:0] = OpQ4:RASPtr[2:0]

// supply old BPT info to decoders for restoring
// if BRCOND Op mispredicted:
SC_OldBPTInfo[14:0] = OpQ4:BPTInfo[14:0]

// supply either fault PC or alternate branch address to
// decoders if BRCOND Op mispredicted:
SC_RestartAddr[31:0] = ExcpAbort ? OpQ5:FaultPC :
   (OpQ4:Emcode ? OpQ4:FaultPC[31:0] :
        BrVecAddr[31:0])
```

continued on next page ...

Pseudo-RTL Description (cont.)

BRCOND Op Resolution Logic

```
// initiate restart if BRCOND Op mispredicted:
BrVec2Emc = SC_Resolve & ~CondVal & OpQ4:Emcode
BrVec2Dec = SC_Resolve & ~CondVal & OpQ4:~Emcode

// remember misprediction:
if (SC_Resolve | SC_Abort)
  SC_MisPred = ~SC_Abort & (~CondVal | SC_MisPred) // enabled flip-flops

// mark BRCOND Op as Completed if correctly predicted:
// enabled flip-flops
if (SC_Resolve & CondVal & BRCOND16) Op16:S3 = 1'b1
if (SC_Resolve & CondVal & BRCOND17) Op17:S3 = 1'b1
if (SC_Resolve & CondVal & BRCOND18) Op18:S3 = 1'b1
if (SC_Resolve & CondVal & BRCOND19) Op19:S3 = 1'b1
```

A BRCOND Op that is being successfully resolved may sit in scheduler OpQuad3 for more than one cycle due to OpQuad4 and OpQuad5 not being able to shift and thus OpQuad4 is not able to shift down. During this time SC_Resolve = 1 and one of the BrVec2XXX signals remains asserted for the entire time, versus for just the first cycle. This is all right since the x86 instruction fetch and decode or OpQuad Sequence fetch areas of the machine which are in the process of being restarted will simply keep on restarting each cycle until the BrVec2XXX signal deasserts. All of the other associated signals such as the vector address will maintain their proper values throughout this time.

Nonabortable RegOp Execution Synchronization Logic

During each cycle, the four Ops within scheduler OpQuad4 are examined for whether any of them is a nonabortable RegOp. If one is found, then it is checked whether there are any preceding mispredicted BRCOND Ops. Due to the OpQuad Sequence coding constraints, any preceding BRCOND Ops must be in a lower scheduler OpQuad (i.e. ,OpQuad5) and thus have all been resolved.

If no such mispredicted BRCOND Ops exist, then the RegOp is allowed to advance into RUX pipeline Stage1. If there is no unexecuted nonabortable RegOp in OpQuad4, but there is a nonabortable RegOp in RUX pipeline Stage0, then the RegOp is unconditionally held up in Stage0. If there is a nonabortable RegOp in OpQuad4 that has not yet executed, but there is no nonabortable RegOp in RUX Stage0, then shifting of

OpQuad4 (and OpQuad5) is inhibited. The pseudo-RTL descriptions that give the equations which describe this logic follow:

Pseudo-RTL Description

Nonabortable RegOp Execution Synchronization Logic

```
NonAbInRUX_0 = (OpInfo_RUX_0(RegOp).Type[5:2] = 4'b1110) & OpV_RUX_0

UnexecNonAbInQ4 = Op16(RU & OpInfo(RegOp).Type[5:2]=4'b1110 & ~S1) |
                  Op17(RU & OpInfo(RegOp).Type[5:2]=4'b1110 & ~S1) |
                  Op18(RU & OpInfo(RegOp).Type[5:2]=4'b1110 & ~S1) |
                  Op19(RU & OpInfo(RegOp).Type[5:2]=4'b1110 & ~S1)

// hold RegOp execution if ...:
NonAbSync = NonAbInRUX_0 & (~UnexecNonAbInQ4 | SC_MisPred | “trap pending”)

// hold Op quad from shifting out of Sched quad 4 if ...:
HoldOpQ4B = UnexecNonAbInQ4
```

Self-Modifying Code Support Logic

Logically, in the scheduler, a detection of self-modifying code is treated as a trap and it factors into the “trap pending” logic. The Store Queue provides the physical address of the store it is preparing to commit. Most of the bits of this address are compared against the instruction address or addresses, if the instructions were from two different (logically consecutive) cache lines of each scheduler OpQuad. If any OpQuad addresses match, then there may be a write to an instruction which has already been fetched, decoded, and is now present in the scheduler—i.e., there must be self-modifying code. Accordingly, the scheduler is then flushed and the fetch/decode process is restarted from the last committed instruction, namely, the modifying instruction.

The following equations describe this functionality. Not all of the address bits need to be compared. A partial-address comparison reduces logic and improves speed while resulting in a very low incidence of “false” matches. In particular, several of the most significant bits and a few least significant bits are not compared. STQ_LinAddr[11:5] are untranslated address bits and thus are the same as STQ_PhysAddr[11:5]. The logic is conceptually only doing a comparison between physical address bits.

PSEUDO-RTL DESCRIPTION

Self-Modifying Code

```
for (i=0; i < 5; ++i) {
    Match1st =
      (STQ_LinAddr[11:5] = OpQi:Smc1stAddr) &
      (STQ_PhysAddr[19:12] = OpQi:Smc1stPg)
    Match2nd =
      (STQ_LinAddr(11:5) = OpQi:Smc2ndAddr) &
      (STQ_PhysAddr(19:12) = OpQi:Smc2ndPg)
    MatchSMC[i] = (Match1st | Match2nd) & OpQi:OpQV
  }

  SmcHit =
    “STQ store is not a special memory access” &
    (“self-modifying code detected” |
     MatchSMC[0] | MatchSMC[1] | MatchSMC[2] |
     MatchSMC[3] | MatchSMC[4]);
```

THE OCU: AN EXPANDED DESCRIPTION

The OCU operates in conjunction with the scheduler and generally operates on the Ops within the bottom two rows of the scheduler. Its principal function is to commit the results of the execution of the Ops within the bottom OpQuad and then to retire the OpQuad from the scheduler. The OCU also handles mispredicted BRCOND Ops and various types of exceptions by initiating abort cycles for them.

Design Note

Committing and Retiring OpQuads (revisited)

One of the differences between committing the results of an Op and the retiring of an OpQuad from the scheduler in the K6 3D is that the actions of commitment and retirement may or may not happen on the same cycle. If some, but not all, of the Ops in an OpQuad can be committed in a given cycle, whatever can be committed is committed. The OpQuad is not retired and removed from the scheduler until all its Ops are committed, so the Ops that were committed will still be in the scheduler. Typically the commitment of all of the Ops in an OpQuad and the retirement of the OpQuad all happen simultaneously.

It is important to note that when result values (both the register value and status flag bits) of an Op are committed, the corresponding byte marks and status modification bits are cleared in the scheduler's Op entry.

There are many types of results or state changes that can stem from the execution of an Op. The principal types of changes are abortable (i.e., they can be speculatively executed and backed out of later) and encompass the following:

1. general register results.
2. status flag results.
3. memory writes.

We include in "general register results" the possibility of partial register modifications and "superset" register dependencies that arise from register forwarding being supported from only one source at a time.[31]

All other state changes are nonabortable (i.e., they cannot be backed out of once executed) and are limited to being the result of RegOp executions. These include changes to:

1. segment registers.
2. non-status EFLAGS bits.
3. special registers, both architectural registers and K6 3D microarchitectural registers.

With respect to general register commitment, the OCU only looks at one OpQuad at a time. It looks at OpQuad5 in the bottom row if it is not all

[31] All needed bytes must come from either one scheduler entry, a result bus, or from the architectural register file.

committed. Otherwise, it looks at OpQuad4 in the second from the bottom row. This is necessary to avoid deadlock situations in which OpQuad3-related or OpQuad4-related synchronization logic is preventing shifts of OpQuad4 and OpQuad5. These situations result from an Op in OpQuad3 or OpQuad4 waiting directly or indirectly for an Op in OpQuad4 to execute or commit before it can execute. Status commitment is always from OpQuad5 (since full forwarding of individual status flag bits is provided). To achieve better performance, store commitment is only loosely coupled with register commitment, (i.e., the OCU can start committing StOps in OpQuad4 while still committing register results from OpQuad5). Given one store commit per clock, this lets the store commit get a head start which is useful when there is more than one StOp in an OpQuad.

In general it is possible for the OCU to commit the state changes of all four Ops in OpQuad5 in one cycle. However, this may take additional cycles. If all the Ops of an OpQuad have been committed or are being successfully committed, then the OpQuad is retired from the scheduler at the end of the current cycle. Otherwise, as many changes as possible are committed during the current cycle and the process is repeated on successive cycles until all changes have been committed.

Design Note

Abortable, Permanent, and Nonabortable State Changes

Abortable state changes are supported by the scheduler and the Store Queue through the general technique of temporarily storing (a) register and status results in the scheduler Op entries and (b) memory write data in Store Queue entries until the associated Ops are committed and retired. Permanent state changes are made during Op commitment when it is safe and definite for the changes to be made. While these new state values reside in the scheduler and the Store Queue, they are forwarded to dependent Ops as necessary. Nonabortable state changes, in contrast, occur immediately during RegOp execution and the responsibility or burden is placed on OpQuad Sequences to ensure sufficient synchronization with surrounding operations.

Commitment Constraints

The commitment of the results of the execution of an Op is constrained by:

1. the Op's execution state—it must be completed.
2. the status of any preceding faultable Ops—these Ops must be completed, which implies that they are fault-free.

3. the status of any preceding conditional branch Op—the associated BRCOND Op's State must be *completed* (versus *unissued*), which implies that it was correctly predicted.

In the case of StOps which generated a memory write there is the additional constraint that only one write can be committed per cycle from the Store Queue into the D-Cache. However, StOps can commit despite preceding not completed RegOps since RegOps can never result in a fault exception.

Design Note

Independent Commitments

The commitment of register results, status flag results, and memory writes are performed independently. For Ops that have multiple results (e.g., a RegOp with both a register result and a status flag result, or a STUPD Op with a register result and a memory write), the various results will not necessarily be committed simultaneously. The commitment of one type of state change can generally get ahead of or behind the commitment of another type of state change. The overall commitment of an Op is considered to occur when the last of all the necessary result commitments associated with that Op occurs. Finally, the commitment of results from multiple Ops is performed without regard to whether the Ops are part of one x86 instruction or separate x86 instructions.

Typically the OCU will commit and retire an OpQuad from the scheduler every cycle. It has the capability to commit up to four register and four status results per cycle and one memory write per cycle. An OpQuad can sit unretired at the bottom of the scheduler for more than one cycle only if it contains multiple memory write StOps or if some of the Ops are sufficiently delayed in their execution that they are not yet completed.

If an Op in the bottom OpQuad needs to be faulted, then all of the succeeding Ops (i.e., Ops higher in the scheduler) are inhibited from being committed. Once all preceding Ops (i.e., Ops lower in the scheduler) within the OpQuad have been committed or are being successfully committed, then the OCU initiates an abort cycle. The abort cycle flushes the entire scheduler and all the execution units of all outstanding Ops.

LDDHA, load default handler address

LDAHA, load alternate handler address

reset OpQuad Sequence

Concurrent with the abort cycle, the OCU vectors the machine to one of two possible OpQuad Sequence entry point addresses—either the OCU default handler address or an OCU alternate handler address; see Figure 2.10 on page 139. The setting of these addresses is supported by the LDDHA Op (LoaD Default Handler Address) and the LDAHA Op (LoaD Alternate Handler Address). Both of these Ops are loaded into the scheduler in a *completed* State and are recognized and "executed" by the OCU when they reach the bottom of the scheduler. The default fault handler address is initialized by the Reset OpQuad Sequence and the alternate handler address is specified by OpQuad Sequences for some instructions and some exception processing cases.

FAULT Ops and LdStOps With Pending Faults

Only certain types of Ops can be faulted, namely LdOps, StOps (except for LEA Ops), and FAULT Ops. For a LdOp or StOp, faults are determined by the second stage of the LU or SU execution pipe respectively. If a fault is detected, the LdStOp is held up in pipe Stage2 indefinitely until either an associated or an unrelated abort cycle flushes it. This results in the characteristic that completed LdStOps are guaranteed fault-free.

The OCU is able to differentiate between a faulting LdStOp and a LdStOp that simply has not yet completed. This is done using signals from the LU and the SU indicating when a faulting Op is stuck in their respective second pipe stages. When the OCU attempts to commit the next uncompleted LdStOp and the associated execution unit is signaling that it contains a faulting Op, then these two Ops must be one and the same. Thus, this Op has encountered a fault. If, instead, the associated execution unit's signal is not asserted, then nothing definite can be determined. In this case, the OCU must continue to wait for the LdStOp to complete.

FAULT Ops are special Ops that are handled somewhat differently by the OCU since they do not execute. They are loaded into the scheduler in an *unissued* state and unconditionally always fault. The handling, though, with respect to commitment and abortion of surrounding Ops is the same as for LdOps and StOps. The *not-completed* state of a FAULT Op is key in causing all of this handling to fall out "naturally"—i.e., without explicit special logic.

Debug Traps and Sequential and Branch Target Limit Violations

In addition to faults on specific Ops, the OCU also recognizes various debug traps. As discussed in the the section titled "Status Flags, Faults, Traps, Interrupts, and Abort Cycles" beginning on page 83, traps are recognized at the end of the instigating instruction. In the case of instructions decoded to an OpQuad Sequence, the traps are accumulated and remembered up until the end of an OpQuad Sequence. Traps are processed on the first OpQuad of the next instruction, which may or may not come along to the OCU in the next clock. This is done so that the FaultPC of the next instruction can be used as the value of the desired TrapPC. Recall that OpQuad sequences end with an "ERET" action field value within the sequencing field of the last OpQuad of the sequence. When such an OpQuad is retired, any accumulated traps are then recorded as now being a pending trap exception or waiting for the first valid OpQuad of the next instruction to come along.

Lastly, the OCU recognizes both sequential and branch target limit violation conditions which, while occurring with just certain Ops within an OpQuad, is associated with the OpQuad as a whole. This is done since the instruction(s) associated with the OpQuad are partially or wholly part of the end of the current code segment limit. If such a violation is detected, it unconditionally causes an abort cycle to be initiated as if a fault was recognized on the first Op within the OpQuad. This bit of OCU functionality handles both sequential and hardware-decoded branch target code segment limit violations.

Aborts for Mispredicted BRCOND Ops

While the OCU is primarily concerned with all the types of Ops that generate abortable state changes, it is also concerned with mispredicted BRCOND Ops. BRCOND Ops are resolved before they reach the bottom of the scheduler and, when mispredictions are detected, the instruction fetch and decode portions of the machine are immediately reset and restarted from the proper instruction address. Therefore, when an OpQuad containing a mispredicted BRCOND Op reaches the bottom of the scheduler and the OCU tries to commit it, the OCU initiates an abort cycle to flush the scheduler and all the execution units of all the older Ops, but does not also restart the upper portion of the processor. This abort cycle also allows new OpQuads to start loading into the scheduler and Ops to immediately be issued.

Aborts for mispredicted BRCOND Ops are similar to aborts for Op faults. For example, for mispredicted BRCOND Ops the commitment of all following Ops is inhibited, pending initiation of an abort cycle. Furthermore, the mispredicted BRCOND Op abort cycle is not initiated until all preceding Ops within the OpQuad, relative to the BRCOND Op, have been committed or are being successfully committed. However, mispredicted BRCOND Op aborts and Op fault aborts are different in that no vectoring to an OpQuad Sequence is initiated for mispredicted BRCOND Op aborts. As mentioned earlier, vectoring to an OpQuad Sequence in the case of a BRCOND Op from an OpQuad Sequence or restarting the x86 instruction fetch and decode in the case of a BRCOND Op hardware decode of a conditional branch instruction has already occurred. If a BRCOND Op is correctly predicted when it reaches the bottom of the scheduler, no action is necessary to "commit" the Op.

The BRCOND Op can be viewed as being either trivially committed or aborted by the OCU—the choice of action is based on the BRCOND Op's scheduler Op entry State. If a BRCOND Op was correctly predicted when it is resolved, its scheduler Op entry State is changed to 'b1111 (effectively completed). However, if it was mispredicted it is left in its initial State of 'b0000. Thus, the prediction status of a BRCOND Op is implied by whether it is completed or not. Treating the state of a BRCOND Op in this way is key in allowing the commitment and abortion of surrounding Ops to occur without explicit special logic.

The Timing of Result Commitments

The actual timing of Op result commitments is relatively simple and can be viewed as happening during the latter part of the commit cycle. In a typical case, an OpQuad reaches the bottom row of the scheduler during some cycle, is committed during that cycle, and is retired from the scheduler at the end of the cycle. During this cycle, while results are being written to the corresponding architectural registers, operand values continue to be forwarded to all dependent Ops from the scheduler (versus from the architectural registers).

Memory Writes

The commit process for memory writes is actually a two-stage process implemented in the form of a two-stage write commit pipeline (see Figure 2.20 on page 169). The first stage of this pipe corresponds to the OCU's commit cycle for a StOp. As far as the OCU is concerned, the StOp has been committed when it enters the second stage of this pipeline (this includes the case of the StOp possibly having been retired from the scheduler). The StOp must enter the second write commit pipe stage before or concurrent with retirement of the associated OpQuad from the scheduler. If a StOp cannot enter this second stage, then the StOp is viewed as not yet being committable and retirement of the OpQuad is held up.

The Timing of Aborts

When the OCU initiates an abort cycle due to an Op fault, the abort signal SC_Abort and its associated OpQuad Sequence vector address are asserted during the commit and retire cycle of the OpQuad containing the faulting Op. During the next cycle the scheduler will have been flushed and the fetch of the first or target OpQuad from the OpQuad Sequence is started. In the case of internal K6 microarchitectural OpQuad Sequences, the scheduler will be empty for exactly this one cycle.

In the case of aborts for mispredicted BRCOND Ops, the abort signal is also asserted during the commit and retire cycle of the associated OpQuad. Since instruction fetch and decode has already been restarted, the scheduler can be reloaded with a new OpQuad as early as the very next cycle. In this case, the scheduler will typically not sit empty for even one cycle.

Design Note

Multiple Ops Within an OpQuad Requiring Abort Cycle Processing

When the OCU recognizes multiple Ops within an OpQuad as requiring abort cycle processing, it chooses the first such Op and initiates appropriate abort actions with respect to that Op, at the appropriate time for that Op. The following Ops requiring an abort are flushed along with all other Ops following the chosen Op.

The following sections detail each aspect of OCU operation. It begins by discussing issues that arise for the various types of Ops that can produce the abortable state changes identified at the beginning of the current section, namely:

1. general register changes produced by all RegOps, LdOps, some StOps (LEA and STUPD), LIMM Ops, and LDK Ops.
2. status flag changes produced by RegOps.
3. memory writes produced by memory-writing StOps.

General Register Commitment

Probably the most obvious function of the OCU is to manage and control the commitment of register result values to the architectural register file. Such values are generated by most types of Ops, e.g., general register changes result from RegOps, LdOps, LIMM Ops, LDKxx Ops, and STUPD StOps. During any given cycle, the OCU examines OpQuad5 and possible OpQuad4 as described earlier in this section to determine which, if any, of the register results can be written into the architectural register file. This is done during the latter part of the cycle via four independent write ports. Each of these writes is performed based on the associated register byte marks, DestBM[2.0], from the appropriate scheduler Op entry. This process applies equally to the architectural registers and to the K6 3D's temporary microarchitectural registers. If an Op is not yet completed and committable, then the associated register file write is inhibited for this cycle.

Design Note

Clearing Byte Marks (Part 1)

If an Op is of a type which conceptually does not generate a register result, then the byte marks will be all clear and the register number possibly undefined. This results in no bytes being modified during the register file write. If t0 is specified as the destination register for an Op, the byte marks will again be all clear. In both of these cases the byte marks were forced to 3'b000 when the Op was loaded into the scheduler. See the K6 3D Design Note, “Clearing Byte Marks (Part 2),” below.

Multiple Simultaneous Full and Partial Writes

In general, when there are multiple enabled file writes, the possibility of contention—in the form of multiple simultaneous writes to the same register—exists. The desired result is that the youngest write succeeds and the other, older writes are inhibited or effectively ignored. Achieving this result is handled within the register file itself, separate from the OCU’s control of the register commitment process. It is based simply on the presented register numbers and associated write enables to the register file.

contention resolution logic

Further, if the contending writes are such that the older writes modify register bytes which are not modified by the youngest write, then the effective register file write must be of the appropriate combination of bytes from each of the possible source Ops. For example, if the first (oldest) Op modifies bytes {3,2,1,0}, the second Op modifies bytes {1,0}, and the third (youngest) Op modifies byte {1}, then the actual register file write takes bytes {3,2} from the first Op, byte {0} from the second Op, and byte {1} from the third Op. This effect is handled locally by the register file’s write control logic. The contention resolution or prioritization logic operates on the basis of individual bytes instead of 32-bit words.

In addition, the nine “match with operand XXsrcY” signals associated with a scheduler Op entry must be forced to indicate no match at the same time that the DestBM bits within that Op entry are about to be cleared (see the section titled “Dynamic Fields OprndMatch_XXsrcY” beginning on page 212. This is due to the pipelined nature of the register operand fetch process within the scheduler. The DestBM bits of an Op entry are used in both stages of this process and must be consistent across both cycles.

Design Note

Clearing Byte Marks (Part 2)

The write enable signals for all four Ops are generated in parallel. For each Op, if it is completed and all preceding Ops are completed (which includes no FAULT OPs and mispredicted BRCOND Ops), and all other "preceding" conditions that can inhibit commitment (e.g., a pending trap exception from the preceding x86 instruction), are inactive, then the associated write enable is asserted. Further, the associated DestBM bits are cleared to reflect the fact that the scheduler entry for this Op no longer needs to provide a register value to dependent Ops. Such values may now be obtained from the register file. Clearing the DestBM field is also necessary in the case of partial register writes since a dependent Op will be held up in a pipe Stage 0 until it can obtain more or all the bytes of the register from the register file if it cannot obtain all its required bytes from this Op. See the K6 3D Design Note, "Clearing Byte Marks (Part 1)," above.

As discussed in the "Register Renaming" section later in this chapter, Op register writes may also take place from OpQuad4 when all the Ops in OpQuad5 have completed. This is accomplished through the use of a 2:1 multiplexer between OpQuad4 and OpQuad5 and by generalizing the RegOp write enable logic to consider either the four Ops in OpQuad5 or the four Ops in OpQuad4. The Ops of the selected OpQuad are renamed OpA through OpD in place of Op0 through Op3 or Op4 through Op7.

Notation

Continuous Numbering of Ops

Sometimes the twenty-four Ops in the scheduler are numbered continuously from Op0 to Op23. Op0 corresponds to the *youngest* Op (i.e., at the top of the scheduler) and Op23 corresponds to the *oldest* Op (i.e., at the bottom of the scheduler). When this is done, Op0 corresponds to OpQuad0[Op3], Op2 to OpQuad0[Op2], Op3 to OpQuad0[Op1], Op4 to OpQuad0[Op0], Op5 to OpQuad1[Op3], Op6 to OpQuad1[Op2], and so on with Op23 corresponding to OpQuad5[Op0]. Note that this numbering scheme is similar to the numbering of OpQuads from OpQuad0 to OpQuad5 and is in contrast to the numbering of Ops within an OpQuad from 0 to 3, where Op0 is the first (and oldest) Op in the OpQuad and Op3 is the last (and youngest) Op in the OpQuad.

The following pseudo-RTL description summarizes the register file write enable equations and the modified DestBM and “match with operand XXsrcY” equations for each Op of the bottom two OpQuads of the scheduler, where Op0 is the oldest Op and Op3 is the youngest Op. These equations ensure the in-order commitment of register results, although not necessarily the simultaneous commitment of these results.

Pseudo-RTL Description

Register File Write Enable

```
RegCmtSel = Op0:S3 & Op1:S3 & Op2:S3 & Op3:S3 &
    (Op0:DestBM = 3'b0) & (Op1:DestBM = 3'b0) &
    (Op2:DestBM = 3'b0) & (Op3:DestBM = 3'b0)

OpA = RegCmtSel ? Op4 : Op0
OpB = RegCmtSel ? Op5 : Op1
OpC = RegCmtSel ? Op6 : Op2
OpD = RegCmtSel ? Op7 : Op3

CmtInh = OpQ5:LimViol | “trap pending”

RegCmtInh = CmtInh | RegCmtSel & (OpQ4:LimViol | ~StCmtSel[2] | SetTrapPend)

WrEnbl0 = ~(RegCmtSel ? OpQ4:LimViol : OpQ5:LimViol) & OpA:S3
WrEnbl1 = ~(RegCmtSel ? OpQ4:LimViol : OpQ5:LimViol) & OpA:S3 & OpB:S3
WrEnbl2 = ~(RegCmtSel ? OpQ4:LimViol : OpQ5:LimViol) &
          OpA:S3 & OpB:S3 & OpC:S3
WrEnbl3 = ~(RegCmtSel ? OpQ4:LimViol : OpQ5:LimViol) &
          OpA:S3 & OpB:S3 & OpC:S3 & OpD:S3

// enabled flip-flops:
if (WrEnbl0)             Op0:DestBM = 3'b0
if (WrEnbl1)             Op1:DestBM = 3'b0
if (WrEnbl2)             Op2:DestBM = 3'b0
if (WrEnbl3)             Op3:DestBM = 3'b0
if (WrEnbl0 & RegCmtSel) Op4:DestBM = 3'b0
if (WrEnbl1 & RegCmtSel) Op5:DestBM = 3'b0
if (WrEnbl2 & RegCmtSel) Op6:DestBM = 3'b0
if (WrEnbl3 & RegCmtSel) Op7:DestBM = 3'b0
```

continued on the next page ...

Pseudo-RTL Description (cont.)

Register File Write Enable

```
// dynamic field flip flops
Op0:“effective match with Operand XXsrcY” =
  Op0:“match with Operand XXsrcY” & ~WrEnbl0
Op1:“effective match with Operand XXsrcY” =
  Op1:"match with Operand XXsrcY” & ~WrEnbl1
Op2:"effective match with Operand XXsrcY” =
  Op2:"match with Operand XXsrcY” & ~WrEnbl2
Op3:"effective match with Operand XXsrcY” =
  Op3:"match with Operand XXsrcY” & ~WrEnbl3
Op4:"effective match with Operand XXsrcY” =
  Op4:"match with Operand XXsrcY” & ~(WrEnbl0 & RegCmtSel)
Op5:"effective match with Operand XXsrcY” =
  Op5:"match with Operand XXsrcY” & ~(WrEnbl1 & RegCmtSel)
Op6:"effective match with Operand XXsrcY” =
  Op6:"match with Operand XXsrcY” & ~(WrEnbl2 & RegCmtSel)
Op7:"effective match with Operand XXsrcY” =
  Op7:"match with Operand XXsrcY” & ~(WrEnbl3 & RegCmtSel)
```

Status Flag Commitment

architectural status flags register

The second function of the OCU is to manage and control the commitment of status flag result values, as generated by status flag modifying RegOps (a.k.a. “.cc” RegOps) to the architectural status flags register. Unlike the commitment of register results, none of the four groups of status results within the bottom OpQuad are written into EFLAGS until the OpQuad is about to be either retired or aborted. In the meantime, full forwarding of individual status flag values is performed as needed. In the normal case, when all the Ops within the OpQuad have been fully committed or are being successfully committed, then the cumulative or overall result of all four status results is written into EFLAGS at the end of the cycle as the OpQuad is retired from the scheduler. In the case of an OpQuad containing a faulting Op or a mispredicted BRCOND Op, only the status results from the Ops before the faulting or BRCOND Op are committed and this cumulative result is written at the end of the abort cycle.

The above process applies to both the architectural status flags and the K6 microarchitectural status flags. In essence, the architectural EFLAGS register is extended to thirty-four bits to make room for the extra two microarchitectural status flags, EZF and ECF. The RDFLG (ReaD FLaG) and WRFLG (WRite FLaG) RegOps reference only the standard 32-bit portion of this extended EFLAGS register.

The generation of the cumulative status result is based on the status bit marks StatMod[3:0] from each of the four Op entries within the bottom scheduler OpQuad (see the section titled "Dynamic Field StatMod[3:0]" beginning on page 211, and the section titled "Dynamic Field StatVal[7:0]" beginning on page 212). As discussed in these sections, the eight x86 status flags are divided into four groups for modification marking purposes instead of having eight individual bit marks. This provides sufficient status modification control within the context of implementing the x86 instruction set architecture. As with updates to a general register within the register file, the possibility of contention exists, i.e., of multiple modifications to the same group of status flags. The desired result, of course, is to take the youngest modification values for each group of status flags as was done for register results.

The generation of the cumulative status result is also based on whether the State of each of the four Ops is *completed* or not. The following pseudo-RTL description summarizes the equation to perform this cumulative result generation or selection process for a status group, which is applied independently for each status group.

No explicit control or constraint on Op commitment and retirement is required for status flag results. Since status flag state changes only results from RegOps and since all RegOps generate register state changes (even if just to microarchitectural register t0), an OpQuad cannot be retired until all RegOps within it are completed and thus also have valid status result values. Consequently, when an OpQuad is ready to retire, it is guaranteed that all status results are available and thus ready to be committed. There is also no need, given the fully unconstrained forwarding of status flag values to BRCOND Ops and "cc-dependent" RegOps, for any clearing of StatMod fields within the Ops of the bottom scheduler OpQuad.

PSEUDO-RTL DESCRIPTION

Status Flag Generation and Selection

```
NextStatFlags[x1..x2] =
  if (Op3:StatMod[x] & Op0:S3 & Op1:S3 & Op2:S3)
    Op3:StatVal[x1..x2]
  elseif (Op2:StatMod[x] & Op0:S3 & Op1:S3)
    Op2:StatVal[x1..x2]
  elseif (Op1:StatMod[x] & Op0:S3)
    Op1:StatVal[x1..x2]
  elseif (Op0:StatMod[x])
    Op0:StatVal[x1..x2]
  else
    StatFlags[x1..x2]
```

STOPS AND MEMORY WRITE COMMITMENT

The third function of the OCU is to commit StOps, particularly StOPs performing actual memory writes. We contrast these types of StOPs with those that do not perform memory writes, such as LEA, CDA, and CIA Ops. LEA Ops do not require any additional commitment handling past commitment of their register results. CIA and CDA Ops do, however, because like normal memory-writing Ops, each of these Ops can result in memory access-related faults. The process of writing data values to "memory"—i.e., to either the D-Cache, the main memory, or the L2-Cache—differs from the commitment of register and status results in a number of ways:

1. StOp commitment involves, in most cases, a memory write and thus an associated store queue entry.
2. at most one memory write can be committed per cycle.
3. the memory write commitment process is a two step process implemented in the form of a two-stage commit pipeline.
4. the OCU looks across the bottom two OpQuads of the scheduler to find StOps with memory writes to commit.
5. the possibility of faults on the associated StOps exists.

When a StOp completes execution, the associated memory address and store data is entered into the store queue. Later, when the memory write of a StOp is committed, this entry is read and retired from the store queue. Since StOps are executed in order and later committed in order, the store queue is managed as a simple FIFO and the matching of store queue entries with associated scheduler StOps is straightforward.

The Commitment Process

The actual commitment process is relatively complicated. We make reference to Figure 2.19 on page 168 to aid in the explanation. A two-step process is required in which the oldest store queue entry is first read and the address looked up in the L1 D-Cache. Then, based on the status of this lookup, the store data is written into the L1 D-Cache or out to memory. In the latter case, the data and address are loaded into the Write Buffer and written out to memory later. From the OCU's perspective the commit process is largely viewed as a single-cycle, single-stage action that either succeeds or is delayed (like the commit process for register and status results). However, it is actually implemented as a two-stage write commit pipeline. The first commit stage C1 corresponds to the commit cycle of register and status results. During this stage no control decisions are made. The L1 D-Cache tag lookup is performed and the accessed tag data is latched for examination during the second commit stage C2. When a write does enter commit stage C2, the associated StOp can be retired from the scheduler and the remainder of the commit process proceeds completely asynchronous to the OCU and the scheduler.

Design Note

One Memory Write Per Cycle

The implication of this "one memory write per cycle" implementation is that OpQuads containing multiple StOps have to sit at the bottom of the scheduler for multiple cycles. In the case of OpQuads containing at most one memory-writing StOp, we have the possible commitment and retirement of an OpQuad each and every cycle, subject to the same sort of constraints that stem from the commitment of register state changes. A corresponding number of cycles is required to commit all the StOps in the OpQuad. For long bursts of StOps, this will typically result in fewer—but still extra—cycles of OpQuads being held up at the bottom of the scheduler before being retired. For short bursts, due to the OCU sometimes being able to get a "head start" by committing StOps from OpQuad4, and due to there sometimes being other retirement delays, there are often times when there are no additional delays due to needing extra cycles to commit the burst of StOps.

The fact that an OpQuad could sit at the bottom of the scheduler for many cycles is partially mitigated by providing support for committing memory writes associated with StOps in the second to bottom scheduler OpQuad as well as the bottom OpQuad. Given that memory writes are committed in order, the OCU can get a "head start" on multiple write OpQuads when the bottom OpQuad does not contain any StOps or when it is held up but otherwise empty of uncommitted memory writes. This helps to better match the OCU's one write per cycle commitment capability to the average number of writes per OpQuad, which is less than one per OpQuad.

Commitment Criteria

During each cycle the OCU's memory write commit logic searches the bottom two scheduler OpQuad entries for the oldest uncommitted memory-writing StOp, i.e., for the next StOp and associated write to try and commit. This selected Op corresponds to the current oldest Store Queue entry. Concurrently, the address of this Store Queue entry is presented to the L1 D-Cache and a tag lookup initiated. The tag lookup is done without consideration of whether the associated StOp is presently committable. If the selected StOp is, in fact, committable, and if this write commit is able to advance into the commit pipe stage C2, then the StOp is considered by the OCU to be committed. In the next cycle the OCU will search for and move on to the next memory-writing StOp.

The criteria for StOp commitment are similar to those for register result commitment:

1. the selected StOp must be completed (in the case of misaligned writes; this also means that both associated Store Queue entries have been created).
2. all older LdStOps within the OpQuad, and possibly the preceding OpQuad if this StOp is in the second to last scheduler OpQuad, must also be completed.
3. there must not be an older mispredicted conditional branch Op.

As mentioned earlier for register commitments, there are other "miscellaneous" conditions as well, but the above serve to illustrate the points that need to be made here. A write commit is able to advance into commit pipeline stage C2 when that stage is either empty or is successfully completing commitment of a write.

If the selected StOp is not committable and this is only because it is not completed, then the OCU examines the signal from the store unit pipeline Stage2 indicating whether a StOp is "stuck" in that stage with a detected fault condition. If there is any such Op, then it is the same StOp as the one being unsuccessfully committed by the OCU, and thus must be aborted by the OCU. An appropriate abort cycle will not be initiated, though, until the StOp is in the bottom scheduler OpQuad, all preceding Ops within the OpQuad have been committed, and there is no preceding mispredicted BRCOND Op. This set of conditions is essentially an extension of the condition for StOp committability. The OCU will remain in this state until an abort cycle is initiated for a preceding Op.

Handling CIA and CDA Ops

While the OCU is primarily concerned with memory-writing StOps, it must also handle CIA and CDA Ops. This is necessary since these Ops generate faultable memory addresses and thus must be examined and committed by the OCU. In the normal case of such an Op having executed fault-free, the OCU trivially spends a cycle on committing the Op and simply moves on to committing the next StOp in the next cycle. Since no store queue entry was created during execution of the Op, no entry is retired from the store queue. If, instead, a fault was detected during execution of the CIA or CDA Op, then it is "stuck" in the store unit pipeline Stage2 and the OCU aborts it in exactly the same fashion as for memory-writing StOps.

Memory References Crossing Alignment Boundaries

The OCU must accommodate the situation that arises when a StOp's memory reference crosses an alignment boundary, eight bytes for eight-byte accesses and four bytes for two-and four-byte accesses.[32] When this occurs, the reference is split by the SU into two memory writes and two associated store queue entries during its execution. In such situations,[33] the OCU takes two cycles to retire the two store queue entries and does not officially commit the StOp until the end of the second cycle. If the StOp has a fault then it must be aborted without retirement of either store queue entry.

The OCU's Write Commit Logic

The following pseudo-RTL description summarizes the functionality of the OCU's write commit logic (where Op0 is the oldest Op and Op3 is the youngest Op in the bottom/last scheduler OpQuad, Op4-Op7 are the corresponding Ops in the second to last scheduler OpQuad, and Op8-Op11 are the corresponding Ops in the third to last scheduler OpQuad).

Its operation is based on a set of CmtMask[7:0] mask bits which represent the OCU's progress in committing memory-writing StOps within the last two scheduler OpQuads. The first several bits, starting from bit 0, are clear, indicating that the OCU has already committed any StOps up to the last such Op position, which contains the next StOp to be committed. All Ops corresponding to the remaining, set, mask bits have yet to be examined for

[32] This irregularity is due to the fact that this is how alignment boundary crossing is defined in the x86 instruction set architecture.

[33] The OCU is able to distinguish aligned versus misaligned StOps because the first Store Queue entry of a misaligned access pair is specially marked as such via a bit that is part of each Store Queue entry.

committable StOps. In addition, from cycle to cycle the OCU maintains a set of UncmtStOp[7:0] bits indicating which Op positions contain uncommitted memory-writing StOps.

During each cycle the OCU selects the next uncommitted StOp and generates a new set of CmtMask mask bits based on the position of this Op. The unmasked Ops (i.e., those with their mask bit set to 0) are examined to determine whether:

1. the selected StOp is presently committable (i.e., is it *completed* and all preceding OPs are fault-free).
2. an abort cycle needs to presently be initiated.

In the former case, if the selected Op is committable and if Stage 2 of the commit pipeline is able to accept a new write commit at the end of the cycle, then the StOp is "committed" and the UncmtStOp bits are updated with new values. The UncmtStOp bits are also updated (shifted) to match any shifting of the last two OpQuads within the scheduler.

Pseudo-RTL Description

Write Commit Logic

```
// StCmtSel = 0000 if OP0 selected (highest priority)
//    "     = 0111 if OP7 selected (lowest priority)
//    "     = 1111 if no Op selected
StCmtSel[3:0] =
  priority_encode((OPQ5:OpQV & UncmtStOp[0]), ... ,
                  (OPQ5:OpQV & UncmtStOp[3]),
                  (OPQ4:OpQV & UncmtStOp[4]), ... ,
                  (OPQ4:OpQV & UncmtStOp[7]))

// this generates a field of zeroes from bit 0 up to and including
// the bit pointed at by StCmtSel[2:0], and a field of ones past
// this up to bit 7
CmtMask[7:0] = {(StCmtSel[2:0] < 3'b111),..., (StCmtSel[2:0] < 3'b000)}

CmtCiaCda =
    (~CmtMask[7] & Op7:Type[2]) |
    (~CmtMask[6] & CmtMask[7] & Op6:Type[2]) |
    (~CmtMask[5] & CmtMask[6] & Op5:Type[2]) |
    (~CmtMask[4] & CmtMask[5] & Op4:Type[2]) |
    (~CmtMask[3] & CmtMask[4] & Op3:Type[2]) |
    (~CmtMask[2] & CmtMask[3] & Op2:Type[2]) |
    (~CmtMask[1] & CmtMask[2] & Op1:Type[2]) |
    (~CmtMask[0] & CmtMask[1] & Op0:Type[2])
```

continued on the next page ...

PSEUDO-RTL DESCRIPTION

Write Commit Logic

```
StCmtInh = CmtInh |
         StCmtSel[2] & (OpQ4:LimViol | SmcHit & ~CmtCiaCda | “trap pending”)

StCmtV = ~StCmtSel[3] & ~StCmtInh &
         (CmtMask[7] | Op7:S3) &
         (CmtMask[6] | Op6:S3 | Op6:RU) &
         (CmtMask[5] | Op5:S3 | Op5:RU) &
         (CmtMask[4] | Op4:S3 | Op4:RU) &
         (CmtMask[3] | Op3:S3 | Op3:RU) &
         (CmtMask[2] | Op2:S3 | Op2:RU) &
         (CmtMask[1] | Op1:S3 | Op1:RU)

Q5StCmtV = ~StCmtSel[2] & ~CmtInh &
           (CmtMask[3] | Op3:S3) &
           (CmtMask[2] | Op2:S3 | Op2:RU) &
           (CmtMask[1] | Op1:S3 | Op1:RU) &
           (CmtMask[0] | Op0:S3 | Op0:RU)

StAdv = ~STQ_FirstAddr & ~DC_HoldSC1 & CHP_AdvSC2 | CmtCiaCda

StRetire = StCmtV & StAdv

Q5StRetire = StAdv & Q5StCmtV

NewUncmtStOp[7:0] = {(CmtMask[7] & Op7:Type=ST), ... ,
                     (CmtMask[0] & Op0:Type=ST)}

// indicates when all memory-writing StOps have been
// committed or are being successfully committed in the
// bottom Sched Op quad

AllStCmt = StCmtSel[2] | Q5StRetire &
           ~NewUncmtStOp[3] &...& ~NewUncmtStOp[0]

// update UncmtStOp bits:
NextUncmtStOp[7:0] = (StRetire) ? NewUncmtStOp[7:0] : UncmtStOp[7:0]
NextUncmtStOp[11:8] =
  {Op11:Type=ST, Op10:Type=ST, Op9:Type=ST, Op8:Type=ST}

// mux followed by a flip-flop:
UncmtStOp[7:4] = LdEntry4 ? NextUncmtStOp[11:8]: NextUncmtStOp[7:4]
UncmtStOp[3:0] = LdEntry5 ? NextUncmtStOp[7:4] : NextUncmtStOp[3:0]
```

continued on the next page ...

Pseudo-RTL Description (cont.)

Write Commit Logic

```
SC_HoldSC1 = ~StQCmtV | CmtCiaCda

StAbort = ~StCmtSel[2] SUViol &
        ((StCmtSel[1:0] == 2'b00) & ~Op0:S3 |
        (StCmtSel[1:0] == 2'b01) & ~Op1:S3 & Op0:S3 |
        (StCmtSel[1:0] == 2'b10) & ~Op2:S3 & Op1:S3 & Op0:S3 |
        (StCmtSel[1:0] == 2'b11) & ~Op3:S3 & Op2:S3 & Op1:S3 & Op0:S3)
```

Memory Read Fault Handling

LdOps normally do not require any special handling by the OCU since they only result in general register state changes. Like most StOps, though, they can also encounter faults during their execution. When this occurs, it is recognized by special logic and handled in the same manner as for StOp faults. To determine whether a faulting LdOp exists in the bottom scheduler OpQuad, the OCU examines each Op in the OpQuad for the following conditions:

1. it is a LdOp.
2. all older Ops are completed and fully committed.
3. there is no preceding mispredicted BRCOND Op.

Again there are other "miscellaneous" conditions as well and, again, the above conditions serve to illustrate the points that need be made here. The conditions identified ensure that all preceding Ops are properly committed. The OCU also examines the signal from the LU pipeline Stage2 indicating whether a LdOp is stuck in that stage with a detected fault condition. At most one of the Ops in the OpQuad satisfies all of these conditions. If one does and the signal from the LU pipeline Stage2 is asserted, then a faulting LdOp is recognized by the OCU and an appropriate abort cycle is initiated immediately to abort this Op and all following Ops. The following pseudo-RTL description summarizes the OCU's LdOp fault handling logic:

Pseudo-RTL Description

LdOp Fault Handling Logic

```
LdAbort = LU2_LUViol &
 (Op0:(Type=LU & ~S3) |
  Op1:(Type=LU & ~S3) & Op0:S3 & ~CmtMask[1] |
  Op2:(Type=LU & ~S3) & Op0:S3 & Op1:S3 & ~CmtMask[3] |  // [3]==[2]!
  Op3:(Type=LU & ~S3) & Op0:S3 & Op1:S3 & Op2:S3 & ~CmtMask[3]
 )
```

FAULT Op Commitment

In addition to commitment of abortable state changes associated with all the normal types of Ops, there are a few special Ops—the FAULT, LDDHA, and LDAHA Ops—that require additional, special commitment handling. None of these Ops are issued to and executed by an execution unit and they have no execution dependencies with other Ops. They are significant only to the OCU.

OpQuad Sequence constraints

The FAULT Op is similar to a faulting LdStOp in that it is handled by the OCU in the same way—an abort cycle is initiated along with vectoring to the current OpQuad Sequence OCU fault handler. Unlike faulting LdStOps, though, there is no problem in determining whether there is a fault to recognize and of when to initiate the abort cycle.

To simplify the OCU's logic for handling FAULT Ops, the following constraints are placed upon OpQuad Sequences:

1. FAULT Ops must be located in the first Op position of an OpQuad.
2. all following Ops in the OpQuad must be NoOps.
3. the next OpQuad to be executed must not contain any memory-writing StOps.

The latter constraints ensure that the OCU's StOp commitment logic can operate blindly of the presence of FAULT Ops, (i.e., without any special consideration). Moreover, the last constraint has no negative effect on performance since the next OpQuad will be unconditionally aborted when the FAULT Op executes, (i.e., the next OpQuad could not have done useful work anyway). In practice, OpQuad Sequences are written so that FAULT OpQuads branch to themselves.

The state of a FAULT Op is initialized to *unissued* when it is loaded into the scheduler. When it reaches the bottom of the scheduler, this inhibits the OCU's OpQuad retirement logic from retiring the containing

OpQuad while the OCU's FAULT Op commit logic immediately initiates an abort cycle. The specifics of this abort cycle are the same as for faults on LdStOps; the only difference is the generation of a unique fault id. The following equation summarizes the OCU's FAULT Op handling logic:

PSEUDO-RTL DESCRIPTION

FAULT Op Handling Logic

```
FltAbort = OpQ5:OpQV & Op0:(Type=SpecOp & (OpInfo(SpecOp).Type=FAULT))
```

LDDHA AND LDAHA OP COMMITMENT

The LDDHA and LDAHA Ops enable OpQuad Sequences to set and to change the OpQuad ROM address to which OCU-recognized exceptions are vectored. The OCU maintains two vector address registers: the first holds a default handler address and the second holds an alternate handler address. The first register is set once by the Reset OpQuad Sequence via an LDDHA Op and is active, by default, for most OpQuad Sequences (for both instructions and exception processing). The second register is set during certain sections of OpQuad Sequences via a LDAHA Op.

For OpQuad Sequences that do not contain an LDAHA Op, any faults recognized by the OCU result in vectoring to the address in the default handler address register. For OpQuad Sequences that contain an LDAHA Op, faults on Ops in OpQuads before the one containing the LDAHA Op still result in vectoring to the default address, while faults on Ops in the OpQuad containing the LDAHA Op or in any following OpQuads up to and including the last OpQuad of the sequence (i.e., the OpQuad containing the ERET), result in vectoring to the address in the alternate handler address register. The retirement of the ERET OpQuad, as well as the occurrence of an abort cycle, reactivates the default handler address register for all following OpQuads, until the next occurrence of a LDAHA Op.

To simplify matters for the OCU, LDDHA and LDAHA Ops are constrained to be located in the first Op position in an OpQuad. Valid Ops are allowed in the following Op positions of the OpQuad (in contrast to FAULT OpQuads where this is not allowed). The following pseudo-RTL descriptions summarize the OCU's LDDHA and LDAHA Op handling logic:

Pseudo-RTL Description

LDDHA and LDAHA Op Handling

```
if (OpQ5:OpQV & Op0:(Type=SpecOp & (OpInfo(SpecOp).Type=LDDHA)))
  DefFltVecAddr[13:0] = Op0:DestVal[13:0]// enabled flip-flOp

LdAltAddr = OpQ5:OpQV & Op0:(Type=SpecOp & (OpInfo(SpecOp).Type=LDAHA))

if (LdAltAddr)
  AltFltVecAddr[13:0] = Op0:DestVal[13:0]// enabled flip-flop

// This implements the requirement for faults on Ops
// within the same Op quad as a LDAHA Op to be vectored
// to the new alternate handler address.
EffAltFltVecAddr[13:0] = (LdAltAddr) ? Op0:DestVal[13:0] :
                                       AltFltVecAddr[13:0]

// OpQ refers to an Op quad field
if (NextOpQ5:Eret & NextOpQ5:OpQV & ~BrAbort | LdAltAddr | ExcpAbort)
  FltVecMode = ~ExcpAbort &
               ~(NextOpQ5:Eret & NextOpQ5:OpQV & ~BrAbort) &
               LdAltAddr// enabled flip-flop

CurFltVecAddr[14:0] =
  (FltVecMode | LdAltAddr) ? EffAltFltVecAddr[14:0] :
                             DefFltVecAddr[14:0]
```

Sequential and Branch Target Segment Limit Violation Handling

In addition to the commitment of state changes associated with each of the Ops within an OpQuad, the OCU also recognizes a special condition tagged with an OpQuad as a whole. Right after the decoders have generated an OpQuad and it has been loaded into the scheduler, if a sequential code segment limit overrun violation is detected, or if a transfer control instruction was just decoded and a code segment limit violation is detected on the target address, the OpQuad is marked to indicate that a code segment limit violation was detected in association with the instruction decode that produced the OpQuad.

When the OpQuad reaches the OCU and is to be committed, the set tag bit (called LimViol) is recognized and an abort cycle is initiated without commitment of any state changes from the Ops within the OpQuad. Effectively the entire OpQuad is faulted. The effect is similar to that which would have occurred if there had been a FAULT Op in the OpQuad. The following equation summarizes the OCU's logic for handling branch target limit violations:

Pseudo-RTL Description

Branch Target Limit Violations

```
LimAbort = OpQ5:(OpQV & LimViol)
```

Mispredicted BRCOND Op Handling

Besides the commitment of abortable state changes and the handling of various special cases, the OCU handles the generation of abort cycles for mispredicted BRCOND Ops. The restart of both the instruction fetch and decode portions of the machine occurs before the BRCOND Op reaches the bottom row of the scheduler (when the branch was resolved as correctly predicted or not, while the BRCOND Op passed through scheduler OpQuad4). The scheduler simply needs to generate an abort cycle and to ensure that only preceding Ops are committed and, as with the generation of abort cycles for Op faults, the abort must not be initiated until all preceding Ops have been committed. The following pseudo-RTL description summarize the OCU's mispredicted BRCOND Op handling logic. The commitment of following Ops is inhibited by the State of the BRCOND Op.

Pseudo-RTL Description

Handling Mispredicted BRCOND Ops

```
BrAbort = Op0:(Type=SpecOp & ~S3) |
   Op1:(Type=SpecOp & ~S3) & Op0:S3 & ~CmtMask[1] |
   Op2:(Type=SpecOp & ~S3) & Op0:S3 & Op1:S3 &
           ~CmtMask[3] | //[3]==[2]!
   Op3:(Type=SpecOp & ~S3) & Op0:S3 & Op1:S3 & Op2:S3
            & ~CmtMask[3]
```

OpQuad Retirement

The OCU retires the bottom OpQuad from the scheduler at the end of the cycle when all of the abortable state changes of the Ops within it have been committed or are being successfully committed. Since this process removes this OpQuad from the scheduler, it also allows the next OpQuad to shift into the bottom row of the scheduler and all earlier OpQuads to shift down as well. During cycles in which not all such Op results have yet been committed, the bottom OpQuad is not retired and either it is retained into the next cycle for further commitment processing or it is invalidated due to an abort cycle. In the latter case, the abort cycle would be in response to some fault having been recognized on one of the Ops within the OpQuad.

DESIGN NOTE

OpQuad Retirement

The retirement of an OpQuad requires that all register results, status results, and memory writes are committed, and that there is no FAULT Op or mispredicted BRCOND Op in the OpQuad. Removal of an OpQuad also immediately occurs if the OpQuad is marked as invalid, a situation taken care of by the scheduler shift control logic.[a] Status results are all committed together in conjunction with retirement or abortion of the OpQuad. Register results are guaranteed to be committed or currently committing if the associated Ops are completed.

[a] See the pseudo-RTL description in the section titled "Dynamic Field Storage Element Operation" beginning on page 190.

The following pseudo-RTL description summarizes the OCU's OpQuad retirement control logic:

PSEUDO-RTL DESCRIPTION

OpQuad Retirement Control

```
OpQRetire = Op3:S3 & Op2:S3 & Op1:S3 & Op0:S3 &
            AllStCmt

if ((OpQRetire | SC_Abort) & ~OpQ5:LimViol)
  StatFlags[7:0] = NewStatFlags[7:0]
  // enabled flip-flop
```

OpQRetire may be asserted for multiple cycles for the same OpQuad. This will occur when shifting of the bottom scheduler entries is inhibited for other unrelated reasons.

ABORT CYCLE GENERATION

The OCU generates abort cycles in two situations, recognition of:

1. an Op fault on a LdOp, StOp or a FAULT Op.
2. a mispredicted BRCOND Op.

Preceding sections have covered the generation of signals initiating an abort cycle: LdAbort, StAbort, FltAbort, LimAbort, and BrAbort. This section describes the generation of the general abort signal[34] and related information.

The Abort signal is simply a combination of all the individual abort signals associated with commitment of specific types of state changes or Ops. The associated OpQuad Sequence vector address, which is used only for fault-related aborts and not BRCOND-related aborts, is simply the currently active fault handler vector address as described earlier.

The Abort signal, itself, flushes the bottom portion of the machine—the scheduler and all execution units—of all outstanding Ops and reinitializes these areas in preparation for receiving fresh new Ops from the upper portion of the machine—the instruction fetch and decode areas. For BRCOND-related aborts this is sufficient since the upper portion of the machine was already restarted earlier by the BRCOND Op resolution scheduler logic.

For exception-related aborts, though, the upper portion of the machine also needs to be restarted at the above OpQuad Sequence vector address. This is accomplished by assertion of the appropriate restart signal SC_Vec2XXX. When the instruction fetch and decode restarts are signaled simultaneously for both a mispredicted BRCOND Op and an Op exception, the latter is given higher priority and the restart vector address and restart signals are generated accordingly.

When a fault-related abort occurs, the OCU also latches information about the fault, namely the x86 instruction fault Program Counter, i.e., the logical address of the x86 instruction associated with the Op being faulted; in other words, the address of the instruction effectively being faulted. The following pseudo-RTL description summarizes the abort cycle generation logic:

[34] SC_EAbort and SC_Abort, where the latter is simply a one-cycle delayed version of the former.

Pseudo-RTL Description

Abort Cycle Generation

```
ExcpAbort = LdAbort | StAbort | FltAbort | LimAbort | TrapAbort | SCReset
SC_EAbort  = ExcpAbort | BrAbort
SC_Abort = SC_EAbort// simple flip-flop

if (TrapAbort)
  FaultId[2:0] = (DTF | SSTF) ? 3'h1 : 3'h0
else if (LimAbort)
  FaultId[2:0] = 3'h2
else
  FaultId[2:0] = LdAbort ? LU2_ViolType : SU2_ViolType

// latch into SR4:
if (ExcpAbort)
  SC_FID[2:0] = FaultId[2:0]// enabled flip-flop
  SC_SR4[31:0] = OPQ5:FaultPC[31:0]// enabled flip-flop

// select OpQuad Sequence vector address:
if (SCReset)
  SC_VecAddr[13:0] = 14'h2200
  ExtEmcVecAddr    = SCExtReset
else
  SC_VecAddr[13:0] =
    (ExcpAbort) ? CurFltVecAddr[13:0] : BrVecAddr[13:0]
  ExtEmcVecAddr =
    (ExcpAbort) ? CurFltVecAddr[14] : BrVecAddr[14]

SC_Vec2ROM = (ExcpAbort | BrVec2Emc) & ~ExtEmcVecAddr
SC_Vec2RAM = (ExcpAbort | BrVec2Emc) &  ExtEmcVecAddr
SC_Vec2Dec = ~ExcpAbort & BrVec2Dec
```

Avoiding Deadlock

There is a difference between committing the results of an Op and retiring an Op from the scheduler. Often these actions happen on the same cycle but this is not always the case. If some, but not all, of the Ops in an OpQuad can be committed in a given cycle, whatever can be committed is committed. The OpQuad cannot be removed from the scheduler until all of its Ops are committed, so committed Ops will still be in the scheduler. When result values (both the register value and status flag bits) of an Op are committed, the corresponding byte marks and status modification bits are cleared in the scheduler's Op entry.

Ops can modify all or just part of a register with a result value (specifically, either of the lower two bytes, both lower bytes, or all four bytes, corresponding to one,-two,-and-four byte size operations). When a scan is being made for the supplier of an operand value required by an Op in pipeline Stage0 or a StOp in SU Stage2), the scan must take into account which bytes of the register are required and which bytes are modified by each of the older Ops that modify the required register. The scan only identifies those Op entries for which there is an overlap between which bytes are needed and which bytes are modified. Destination byte mark (DestBM[2:0]) bits keep track of which parts of a result register are modified by an Op. The result of the OCU clearing these bits during Op commitment is that the scan logic realizes that this Op no longer looks like it modifies the required register and the particular bytes, if needed, can be supplied by the architectural register file.

Why is it that the OCU needs to be able to commit some of the Ops in an OpQuad even though it cannot commit all of them? This approach is in contrast to simply waiting until you can commit all of the Ops in the OpQuad before committing any of them—i.e., commit and retire all of them at the same time. The answer has to do with avoiding potential deadlock situations.

Suppose that one of the Ops in OpQuad3 or OpQuad4 cannot get its source operands and, because it is a cc-dependent regOp, a nonabortable RegOp, or a BRCOND Op, this causes shifting of this OpQuad and the one or two below it to be inhibited as discussed earlier. If an older Op, say in OpQuad4, is producing some of the required register operand value, but not all of it, the scheduler needs to wait for the OCU to commit the older Op. But, if it cannot get to the bottom of the scheduler to be committed because the scheduler cannot shift, then the scheduler and OCU are deadlocked.

For example, suppose the Op that gets stuck in OpQuad3 or OpQuad4 requires all four bytes of the AX register as one of its source operands. Let's assume that two of these bytes are modified by an Op lower down in the scheduler, one of the remaining required bytes is modified by a different Op lower down in the scheduler, and the final required byte is in the architecture register file. The K6 does not support being able to source or forward some of the bytes from one Op entry and some of the bytes from another Op entry and some of the bytes from the architectural register file. In general, all required bytes must be forwarded from just one source. In this case, the scheduler will have to wait until the OCU can commit to two Ops lower down in the scheduler, so that all of the required bytes can be obtained from the architectural register file. But, since the scheduler cannot shift, these Ops will never get committed.

Another, but more subtle example, is the deadlock that might arise within a single OpQuad. even if it is already in OpQuad5 of the scheduler.

Suppose one of the later Ops in the OpQuad is partially dependent on the results of an earlier Op in the same OpQuad. Unless the OCU can commit the earlier Op so that the required operand can be obtained from the architectural register file, the dependent Op will never be able to execute.

To resolve such potential deadlock situations, the OCU is able to commit whichever Ops in an OpQuad that can be committed, instead of having to wait for all of the Ops to be able to be committed. Further, the OCU is able to commit Ops from both OpQuad4 and OpQuad5 (the bottom two rows of the scheduler).

Since these deadlock situations arose from the fact that the scheduler cannot provide a set of required register bytes from a collection of sources—if four bytes of a register are required, the scheduler cannot get one byte from one place, one byte another place, and the other two bytes from a third place—an alternative solution would be to provide the resources to allow the scheduler to do this. This approach, however, proves to be much more costly (somewhat in speed and very much in size) than providing the OCU the flexibility just described.

Register Renaming

Programs are written under the assumption that their constituent instructions will execute sequentially. For architectures that support the out-of-order execution of instructions, this assumption is not true and certain conflicts may arise that need to be avoided to ensure proper program behavior. Since the issues discussed in this introductory section are applicable to the x86 instruction set architecture and the K6 3D microarchitecture, the term *command* is used in this section to mean either an *x86 instruction* or a *RISC86 operation*. In general, a command either:

1. requires and “reads” zero, one, or more operand values to be operated upon from some form of storage such as registers or memory.
2. produces one or more result values and “writes” them into some form of storage such as registers or memory.

There are potential conflicts between various combinations of reading operand values and writing result values from and to common storage locations when they are allowed to occur out of their sequential program order. There are four basic combinations of reading and writing of operand and results values from and to the registers and memory locations for two commands.

Table 3.11 Potential Conflicts When Reading/Writing Operand/Result Values

1st command	2nd command	Notation or Terminology	Example of Potential Conflict
Read	Read	RAR Read After Read	In this case, both commands use the same result, produced by some previous command. There is no conflict and the two commands can be executed in any order.
Read	Write	WAR Write After Read	In this case, the 1st command reads a result produced by some previous command from a specific storage location and then the 2nd command writes to the same storage location. A conflict exists if the 2nd command writes a new result to the storage location in question before the 1st command reads the older, previous result value from it.
Write	Read	RAW Read After Write	In this case, the 2nd command uses the result produced by the 1st command. A conflict exists if the 2nd command reads the result's storage location before the 1st command has written its result to the storage location in question as the 2nd command will be reading the wrong result value—the result produced by some other command.
Write	Write	WAW Write After Write	In this case, the 1st command writes its result value to a specific storage location and then the 2nd command writes its result to the same storage location. A conflict exists if the 2nd command's write occurs before the 1st command's write as the storage location is left with the wrong result. Subsequent reads to the storage location will read this wrong value.

Implications for Pipeline Operation

Pipeline operation, can be affected by the presence of these conflicts as execution will be stalled until such conflicts, when they exist, are resolved. As Hennessy and Patterson point out in the last reference cited in the following "Historical Comment and Suggested Readings" inset, commands that use registers and have a potential Write-After-Read conflict or a potential Write-After-Write conflict can execute out of order if the registers are renamed (see Section 4.1 of the Hennessy and Patterson text). It is therefore relatively important that a high-performance method of supporting register naming be an integral part of the processor's core.

Historical Comment and Suggested Readings

Pipeline Design

There is a range of terminology used to describe the conflicts that arise in the design of pipelines and the techniques used in both processor design and compilers to resolve them. They have been called, among other things, "dependencies" or "hazards" in various architecture-and compiler-related work. For a treatment of the issues from an architecture point of view, see Peter M. Kogge's book, *The Architecture of Pipelined Computers*, McGraw-Hill Book Company, 1981, and Harold Stone's book, *High-Performance Computer Architecture*, 3rd Edition, Addison-Wesley Publishing Company, 1993. Kogge's book gives a rich history of the resolution of these conflicts in pipelined computers. His discussions of Thorton's scoreboarding technique used in the CDC6600 and Tomasulo's algorithm used in the IBM 360/91 are quite interesting.

For a compiler-related treatment of these issues see: "Local Microcode Compaction Techniques," Dave Landskov, Scott Davidson, Bruce Shriver, and Pat Mallet, *ACM Computing Surveys*, Vol. 12, No 3, September, 1980; "Microcode Compaction: Extending the Boundaries," Dave Landskov, Josh Fisher and Bruce Shriver, *International Journal of Computer and Information Sciences,* Vol. 13., No. 1, February, 1984; "Microcode Compaction: Looking Backward and Looking Forward," Josh Fisher, David Landskov and Bruce Shriver, *Proceedings of the National Computer Conference,* NCC '81, AFIPS Press, Chicago, Illinois, May, 1981.

Since pipelining is typically an integral part of current microarchitectures, there is an extensive treatment of the issues involved in classifying and resolving these conflicts in this literature as well. See, for example, the article by Wen-Mei Hwu, Richard E. Hank, David M. Gallagher, Scott A. Mahlke, Daniel M. Lavery, Grant E. Haab, John C. Glyllenhaal and David I. August, "Compiler Technology for Future Microprocessors," *Proceedings of the IEEE*, Vol. 83, No. 12, 1995, pp. 1625-1640. You can find the full text version of this article on the CD-ROM. See also Mike Johnson's, *Superscalar Microprocessor Design*, Prentice-Hall, 1991; and John Hennessy and Dave Patterson's *Computer Architecture: A Quantitative Approach,* 2nd Edition, Morgan Kaufmann Publishers, Inc., 1996.

An architecture's instruction set accesses a set of registers that are used for storing values associated with general registers, status flags, and other architectural state-related information. This set of registers is often called the architectural register set or architectural register file[35] (see the section titled "Architectural and Microarchitectural Registers" beginning on page 88). The values stored in it at any instant in time are called the architectural machine state or instruction set architecture machine state. The microarchitecture typically

[35] Although separate register files are often used for each of the various types of state, e.g., a separate register in the case of status flags distinct from the architectural register file for general registers.

has an additional number of registers used to store additional microarchitectural machine state, (i.e., operand values, status flags, and state information that is used exclusively in the microarchitecture and not explicitly visible to the instruction set architecture). Further, the microarchitecture typically has a different number of physical registers, most often a larger number, that are used to store uncommitted as well as committed values in these architecture and microarchitecture registers. Before proceeding in this section, we recommend you consider the following review:

Suggested Review

It might be useful at this point for you to review the following sections in Chapter 2 that are particularly relevant to the issues in register renaming: the section titled "Architectural and Microarchitectural Registers" beginning on page 88, the section titled "Register Number and Name Mappings" beginning on page 91, and the section titled "Special Registers and Model Specific Registers" beginning on page 94.

From the sections identified in the above "Suggested Review" inset, we recall that the K6 3D has twenty-four 32-bit integer registers in the integer architectural register file. The K6 3D also has twenty-four 32-bit integer renaming registers. The twenty-four integer registers in the integer architectural register file consist of eight registers that correspond to the x86 architecture 32-bit-general purpose registers (EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI) and sixteen microarchitecture scratch registers (t0 through t15). The twenty-four renaming registers are located in the scheduler's twenty-four Op entries—one per entry. The K6 3D also has nine MMX/3D 64-bit architecture registers and twelve MMX/3D 64-bit renaming registers. The nine architectural registers consist of eight that correspond to the x86 architecture MMX 64-bit registers (MM0 through MM7) and one microarchitecture scratch 64-bit register (MMt1).

Register mapping is the process of associating one set of registers with another set of registers. The mapping can be static (bound before execution) or dynamic (done at execution time). If the process is dynamic, (i.e., "renaming" (re-mapping) occurs during execution, it is called register renaming.

In the K6 3D, both the x86 architectural registers (e.g., AX-DI, MM0-MM7, CF, ZF, SF, OF, PF, and AF) and the microarchitectural registers (e.g., t0-t15, MMt1, ECF, and EZF) are renamed to physical registers. Thus, in the context of the K6 3D, register mapping is the process of associating the set of x86 architectural registers and the set of K6 3D microarchitectural registers with specific physical registers which actually store the register values. As this process is dynamic, it is appropriate to call this process register renaming. The renaming is such that each x86 architectural register and each K6 3D microarchitectural register having a valid value has a corresponding physical register mapped to it.

Register renaming is often accomplished through the use of tags that are used to identify the various registers. The tags can be thought of as register numbers or register identifiers. When an architectural or microarchitectural register identifier is presented to the mapping mechanism (e.g., a mapping table), the current corresponding physical register identifier is output. We will discuss two types of register renaming schemes, explicit and implicit.The mappings must be complete, i.e., each architectural register having a valid value must have a corresponding microarchitectural register mapped to it at each point in time when a valid value is associated with the architectural register.

Explicit Register Renaming

Some architectures implement an explicit register renaming scheme. In such schemes there is an explicit mapping or translation of architectural registers into physical registers. These schemes employ a translation or mapping mechanism that:

1. maintains a list of which physical register numbers are currently not in use and thus available to be allocated.
2. assigns a physical register number to be associated with an architectural number; results that are to be held in the architectural register are held in the associated physical register.
3. produces or outputs the physical register number associated with a given architectural register number.

Such schemes often require additional information such as an indication of the validity of the data value a register contains. A copy of the mapping information is typically required to be able to restore the machine state when an exception, abort, or mispredicted branch is encountered. We will now discuss two basic approaches that are used to implement explicit renaming schemes.

Suggested Review

It might be useful at this point for you to review several pipeline diagrams shown in Chapter 2 (Figure 2.12 on page 159, Figure 2.14 on page 163, Figure 2.15 on page 164, Figure 2.18 on page 167, and Figure 2.21 on page 175) and the text that accompanies them, as well as the scheduler diagram Figure 2.9 on page 130 and its related discussion.

Using One Pool of Registers

One type of approach used in explicit register renaming schemes is to have a general pool of physical registers that can be used to hold both committed and uncommitted values in the registers of the architectural/microarchitectural register set. A tag associated with each of the physical registers points to the specific architectural or microarchitectural register that it is currently assigned to. Similarly, there is a tag associated with each architectural or microarchitectural register pointing to the physical register that it is mapped to. At a specific instant in the instruction stream, the set of values of all such tags is the *current mapping* of the architectural and microarchitectural register numbers onto the physical register numbers. This means that the physical registers identified hold the latest or current values for the architectural and microarchitectural registers identified. At that specific instant in time, some of the other physical registers may be free while others might be holding older values of various architectural and microarchitectural registers.

The tags are located in a centralized resource, say a mapping register or renaming table. Copies of older versions of mapping information are also required to be able to restore the machine state when an exception or a mispredicted branch is encountered. If there is a 1-to-1 correspondence between each operation and a corresponding register modification (i.e., each operation modifies at most one register), the number of physical registers required is roughly (X + Y + Z) where:

1. X is the number of architectural registers.
2. Y is the number of microarchitectural; registers.
3. Z is the number of speculative copies of architectural and microarchitectural register values (i.e. computed register values that are not yet committed).

In the explicit mapping approach, when instructions are decoded the architectural register identifiers or numbers used in the instructions must be translated to the current corresponding physical register identifiers. The current corresponding physical register numbers are used in the internal operations associated with the decoded instructions. The mapping information is modified in the decoding process and in the commitment process. For example, assume the following instruction is decoded:

```
AX <— AX + BX
```

(i.e., the value contained in AX is added to the value contained in BX and the result replaces the original value contained in the AX). Assume that the mapping information indicates that the current values of the architectural registers AX and BX are in physical *Register 3* and *Register 7*, respectively.

Assume that physical *Register 24* is currently free and unused. During the decode process a physical register is allocated to hold the result of the add instruction so that the previous value of AX is not immediately lost when the operation is speculatively executed. In this example, physical *Register 24* is assigned to hold the result of the addition and the mapping information is modified to reflect this assignment. The resulting operation will read physical *Register 3* and *Register 7* for its two source operands and then put the result of the addition into physical *Register 24*.

At this point, the most recent value of AX is held in physical *Register 24* and its previous value is held in physical *Register 3*. The next instruction that references AX will read from physical *Register 24* and not from physical *Register 3* as the mapping information has been modified to reflect this. When a mispredicted branch or an exception occurs, all following instructions that have been decoded will need to be flushed out of the machine. Correspondingly, the mapping information needs to be restored to a set of mappings corresponding to the point where execution will be restarted. If the example instruction had been executed speculatively and a mispredicted branch occurred such that the instruction should not have been executed, the mapping information would have to be restored to reflect that the current value for AX is in physical *Register 3* and not physical *Register 24*. When a register is committed (i.e., its results are made permanent in the architectural register file), the mapping information needs to reflect that the most recent value of the register is the current value and any registers that were holding older values are now free to be reallocated as new instructions are decoded.

Using Two Pools of Registers

A variation of the approach just described is now given. Instead of having just one general pool of physical registers you might have two pools. One pool is a set of registers which is used as a *committed architectural/microarchitectural* register file. The other set is used to hold register values until they are committed, i.e., an *uncommitted register set*.[36] As with the preceding approach, architectural register numbers are converted into physical register numbers at decode time and then the mapping information is updated. Operations can reference either (or both) the committed and uncommitted register files.

When a register value is committed, the value needs to be written from the uncommitted register file to the committed register file. In terms of the previous example, after the add operation executed, *Register 24* in the uncommitted register file was holding the new value of AX. At the time

[36] The term *register set* is used to reflect the fact that these registers do not form a register file in the conventional sense of one register per register address.

of commitment, the value from this register is written to the AX register in the committed architectural register file. At this point, *Register 24* in the uncommitted register file is also freed up to get reallocated as new instructions are decoded.

Implicit Register Renaming

In contrast to such explicit renaming schemes, the K6 uses an implicit register renaming scheme. This implicit scheme is similar to the immediately preceding scheme (*Using Two Pools of Registers*) in having two pools of physical registers, committed and uncommitted register files. We will first explain how this is done for integer instructions and then for MMX and 3D instructions.

The scheduler uses forty-eight physical registers when processing the (up to) twenty-four Ops that can be in the scheduler at any point in time. The registers consist of two register groups, twenty-four committed state registers and twenty-four renaming registers. The twenty-four general registers consist of eight registers that correspond to the x86 general-purpose registers—(i.e., EAX, EBX, ECX, EDX, EBP, ESP, ESI and EDI), and sixteen microarchitectural scratch registers for use within OpQuad Sequences. These twenty-four registers are located in the Architectural Register File, shown in Figure 2.2 on page 69. The twenty-four renaming registers correspond to the twenty-four DestVal fields in the scheduler, one DestVal field per scheduler Op entry as discussed in the section titled "The DestVal field plays an important role in the K6's implicit renaming strategy. The OpQuad Expansion Logic circuitry used to initialize the DestVal field and the scheduler circuitry logic associated with dynamic field DestVal is given in the following pseudo-RTL description:" beginning on page 210. The renaming registers hold result register values while they are not yet committed. We repeat, for your convenience, part of a "Historical Comment and Suggested Reading" inset titled Reorder Buffer on page 134:

Excerpt from an Earlier
Historical Comment and Suggested Reading

Reorder Buffer

Processors that support speculative and out-of-order execution typically have operations completing execution before they are ready to be committed. The results of such operations are not committed (i.e., producing permanent state change) until it is safe to do so. The collection of storage elements that holds the results of the as-yet-uncommitted operations is often called a reorder buffer, for it is from this buffer that the instructions which have been executed out of order will be committed in an in-order fashion. A reorder buffer also supports the use and forwarding of results of completed operations as source operands for other dependent operations. The K6 is an example of a microprocessor in which its reorder buffer (i.e., included in the scheduler's centralized buffer functionality) also serves as an environment to support register renaming. Its renaming registers hold result register values until they are committed. It holds these values in the DestVal field of the appropriate Op entries in the scheduler.

As seen in the first portion of this chapter, each scheduler Op entry holds one RISC86 operation which can modify at most one register. At this point, in contrast to earlier sections, we are talking about general registers, i.e., excluding "status flag" registers. Fields Src1Reg, Src2Reg, and SrcStReg in the Op entry hold the register numbers identifying the registers for the first source operand Src1, the second source operand Src2, and the store data operand (which exists only for StOps) of the Op. The register result of the operation is stored in the Op entry's 32-bit DestVal field.[37] The DestVal field is effectively the Op entry's register result renaming register. The architectural register identity of the renaming register within the Op is specified by value in the DestReg field of that Op (see the section titled "Static Field DestReg[4:0]" beginning on page 201). The DestVal field could also be called the *local implicit renaming register*. Since there can be up to twenty-four Ops outstanding in the scheduler at any time, each having a DestVal field, the K6 has twenty-four local implicit renaming registers.

[37] This is for the integer case. The MMX and 3D cases will be examined shortly.

The Basic Scheme

As was seen earlier in this chapter, when Ops are committed the value in the Op entry's local renaming register is written into the architectural register file, as was done in the *Using Two Pools of Registers* explicit renaming schemes just discussed. The K6 3D design exploits the following features of the microarchitecture that were discussed in Chapter 2:

1. the 3-bit architectural register numbers are trivially converted to 5-bit microarchitectural register numbers effectively implementing a fixed mapping between x86 registers and eight corresponding K6 microarchitectural registers.[38]
2. the K6 has twenty-four microarchitectural registers (eight corresponding to x86 architectural registers) and the others are scratch registers (for use within instruction OpQuad sequences).
3. the twenty-four microarchitectural registers are renamed using forty-eight physical registers.

The implicit renaming scheme is based on the fact that the scheduler contains a physically ordered list of Op entries and thus a physically ordered list of locally implicit renaming registers. Renaming is achieved by dynamically determining from where the required source register operand values are supplied. These values can be supplied from scheduler Op entries or, by default, the architectural/microarchitectural register file. Starting from the Op entry that requires a register operand value, a scan is made down the scheduler toward the bottom row of the scheduler (i.e., toward the older Op entries) looking for the first Op that modifies the required register *and* the required bytes of that register. Essentially, the scan compares the value in the Op's Src1Reg, Src2Reg, or SrcStReg field, as appropriate, with the value in the DestReg field of the older Ops. If a match occurs, (i.e., such an Op is found), it will be the supplier of the operand value, as it is the most recent older modifier of the required register. If such an Op is not found, then by default the architecture/microarchitecture register file is the supplier of the required register operand value.

Assume an Op is found. If the State field of that Op entry indicates the Op has already completed execution, then the value in that Op entry's DestVal field (i.e., its renaming register) can be read out onto the appropriate operand bus and used as the required source operand value. If instead the State field indicates that the Op is currently completing execution, then it is possible to "bypass" the desired operand value off of the appropriate result bus. If the State field indicates that the Op entry has not

[38] See the section titled "Architectural and Microarchitectural Registers" beginning on page 88.

yet started execution or is not finished completing execution, then an operand value is not yet available. In any case either:

1. during the operand scan/selection process, an Op is identified as a source or supplier for each of the required source operand register values.
2. if the scan cannot find any older relevant Op entries, then by default the architectural/microarchitectural register file is the supplier of the required register operand value. This means that what is used is, in fact, the oldest and committed value for the required register.

Once the register operand values are available and supplied and the appropriate execution unit is available, the Op is executed and the register result value gets loaded into the Op entry's DestVal field.

The MMX and 3D Registers

As mentioned earlier, the K6 3D has twenty-four 32-bit integer registers in the architectural/microarchitectural register file and twenty-four 32-bit integer renaming registers. The latter registers correspond to the twenty-four Op entries within the scheduler.

The MMX/3D instructions operate on 64-bit values and share usage of the eight architectural MMX registers. In contrast there are nine 64-bit MMX/3D registers in the architectural/microarchitectural register file and twelve 64-bit renaming registers. The nine architectural registers consist of eight that correspond to the x86 architecture MMX registers, plus one microarchitectural scratch register. Before indicating where the twelve renaming registers are located, let's first discuss why there are twelve of them.

Recall from Figure 2.3 on page 81 that the RUX and RUY can execute any combination of MMX and/or 3D instructions that do not involve the simultaneous use of the same shared execution logic in the two pipelines. Further, the decoders can decode up to two MMX or 3D instructions per cycle. These instructions each produce zero or one MMX or 3D RegOps for a maximum possible of two MMX/3D RegOps in an OpQuad. Similarly, each instruction may produce up to one LdOp. From an architectural perspective, each instruction can produce Ops with at most one register result. Consequently, each instruction, although able to produce up to two Ops, only requires up to one MMX/3D renaming register. (The case of a "LdOp;RegOp" combination is finessed by taking advantage of the fact that the LdOp register result is only used by the RegOp and that the two Ops are guaranteed to execute in sequential order. This allows one physical renaming register to be used by both Ops without actual conflict.)

If two MMX/3D instructions are decoded, the first pair of Ops in the OpQuad corresponds to the first MMX or 3D instruction and the second pair of Ops in the OpQuad corresponds to the second MMX or 3D instruction. Since the K6 3D requires only one MMX/3D renaming register per instruction, the scheduler contains one MMX/3D renaming register for the first pair of Ops in an OpQuad and a second MMX/3D renaming register for the second pair of Ops in an OpQuad. Given that there are six OpQuads in the scheduler, this means a total of twelve MMX/3D renaming registers are needed.

In the basic implicit renaming scheme just discussed, the scan for source operands examines the DestReg fields to look for a match with each of the SrcReg fields in the Op that require a source operand. The MMX/3D renaming registers are located within the scheduler itself according to the following algorithm. There is a 32-bit DestVal per Op entry and a 64-bit MDestVal per pair of Op entries. There is no physical sharing or reuse of the integer DestVal fields to hold MMX/3D register values.

But, how are these registers identified, i.e., what are their register numbers? Recall that the static fields Src1Reg[4:0], Src2Reg[4:0], and SrcStReg[4:0] hold the register numbers that identify registers which respectively hold the first source operand Src1, the second source operand Src2, and the store data operand of an Op, while the static field DestReg[4:0] holds a register number identifying the destination register of the Op. Each of the register number fields are five bits wide which means that thirty-two microarchitectural registers can be uniquely identified. A design decision was made that of the thirty-two possible registers, twenty-four of them were to be used for the twenty-four integer (micro)architectural registers and that the remaining eight would be used to identify eight of the nine MMX/3D architectural registers. The register number for the ninth register, the MMX/3D scratch register MMt1, was chosen to be the same register number as that of the integer register t1. This results in the constraint that MMt1 and t1 cannot be used in such a way that both are "live" at the same time, holding both an integer value and an MMX value for use by following integer and MMX/3D Ops.

The Basic Implicit Renaming Scheme Revisited

The MMX and 3D instructions do not deal with partial register modifications as do many of the x86 instructions. The bottom line is that the same exact scan process (and, in fact, the exact same logic) is used in renaming the MMX and 3D registers as was described earlier, taking into account that partial register modifications cannot occur. A scan to locate an MMX/3D operand will result in the 64-bit operand value located in the "MDestVal field" of two adjoining Op entries being driven onto the appropriate operand bus if a match is made on the DestReg field of either

Op entry. If no match results, the operand value is located in the MMX/3D architectural register file.

The handling of operand scan and selection and forwarding for integer and MMX/3D register values is unified within the scheduler—the same scheme and, in fact, the same scheduler logic and generated control signals function for handling both. This is enabled by having the twenty-four integer and nine MMX/3D registers use the same 5-bit register identifier space and by the appropriate setting of fields (such as the source and destination byte marks), and by the pair-wise OR'ing of per-Op-generated DestVal read/write/etc. control signals to produce the associated per pair of Ops MDestVal control signals.

Design Note

FPU Register Renaming

The K6 renaming scheme does not apply to the floating-point unit. As discussed in the section titled "The Execution Units" beginning on page 77, the floating-point unit is essentially the core of the FPU from the Nx586 and is considered a "mini" microprocessor with a simple interface to the scheduler. It has its own Op queue, floating-point Op decoder, register file and register renaming scheme, control logic, dependency checking, and abort handling.

Differences Between the Implicit and Explicit Register Renaming Schemes

There are a number of different issues that can form the basis of discussing differences between the implicit and explicit register renaming schemes. Among them are: the complexity of the solution, dependency checking, operand forwarding, dealing with partial register modifications, the amount of registers and logic required, and the scalability and flexibility of the scheme. There are a variety of approaches taken to implement renaming schemes. To do a good classification would be difficult because of the number of variables involved; however, some coarse comparisons can be made.

Complexity of the Solution

Let's consider what has to be done in any renaming scheme somewhere around instruction decode time, or shortly thereafter, if multiple decodes are supported. Multiple decodes means, in general, that multiple register renamings will be required. Since there may be data dependencies among instructions, the multiple renamings may be interdependent. If an explicit renaming scheme is being used, some mechanism is required to reflect these dependencies in the mapping information and to accommodate them during the multiple decode process itself as well as during an abort cycle. No such mechanism is required in the K6 3D's implicit renaming scheme.

On the other hand, when it comes time to actually obtain an operand, explicit schemes are somewhat simpler. If *Register 5* is needed, there's only one other Op in the machine that can be modifying *Register 5*. In an explicit renaming scheme, each older modifier of an architectural register, say AX, looks like it is modifying a different physical register. Suppose the value in AX is required. In the explicit scheme where there is one pool of registers, there is one physical register mapped into the AX register at any point in time, say *Register 5*, and that is the required register. That value may be a committed value or not but there's only one such required register. In the two pools scheme, either there is some register that contains a new value for the AX register, say *Register 7* (and there is only one of these at most) or, if no such register exists,[39] then the value for AX must be in the architectural register file. In either scheme, there is only one supplier for the required value and there is no need to distinguish between potentially many modifiers of a register as is required in the K6 3D.

The K6's implicit scheme is realized within the framework of a centralized scheduler buffer, not within a framework of distributed reservation stations. It is not clear how easily it extends to a distributed environment. Explicit schemes are, however, implemented in both centralized and distributed instruction window environments. Also note that handling of partial register results is relatively straightforward with the K6 3D's implicit scheme. This can be trickier and more involved with explicit schemes.

Resources Required

In an explicit scheme there needs to be some mechanism for keeping track of which physical registers are free and which are mapped to specific architectural registers for holding speculative, uncommitted results and what those mappings are. Multiple copies of such mappings need to be retained to accommodate aborts back to the architectural register state corresponding to just before or after any of the recent, still speculatively executed instructions. The specifics of what is needed depend, of course, on the particular explicit scheme employed and its implementation. In the K6, the renaming registers are contained within the scheduler Op entries. Therefore, no lists need to be maintained to identify free registers. The renaming registers are available to Ops as they are loaded into the first row

[39] That is, neither Register 7 nor any other register holds a value for AX.

of the scheduler. Further, no mappings or multiple copies of mappings are required due to the way aborts are handled.

The number of registers required to do register renaming is basically identical for both approaches. As identified earlier, a total X+Y+Z registers are required. In the explicit scheme, additional registers are required for the various copies of the mapping information (current table and history tables) and logic to do the mapping, multiple decodes, abort cycle handling, etc. In the centralized buffer, implicit scheme, no additional information is required for such mapping information. Logic is required to do the scan, but no special logic is required to handle multiple decodes or abort cycles.

Summary of the Chapter

A detailed examination of three main aspects of the microarchitecture of the K6 3D in more detail—its scheduler, its operation commit unit, and its register renaming scheme—has been given in this chapter. As specific microarchitectural concepts were introduced, pseudo-RTL descriptions were given for typical chunks of logic that could be used to implement these concepts. Hopefully, the combination of "diagram, text, and pesudo-RTL descriptions," augmented by independent simulations by the reader, helped bring about an understanding how a contemporary superscalar microprocessor—with its multiple execution units, predecode logic, multiple decoders, scheduler, operation sequences, branch resolution logic, operation commit unit, register renaming scheme, and on-chip L1-Cahce and L2-Cache—might be designed and implemented. We intended Chapters 1 and 2 to provide a detailed and coherent context for the reader to study and understand microarchitecture elements and their impact on the overall design of a microprocessor. We now begin the second part of this book in which we attempt to provide a detailed and coherent treatment for understanding a wide range of platform-related and systems-related issues.