EECS 252 Graduate Computer Architecture

Lec 9 – Precise Exceptions

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

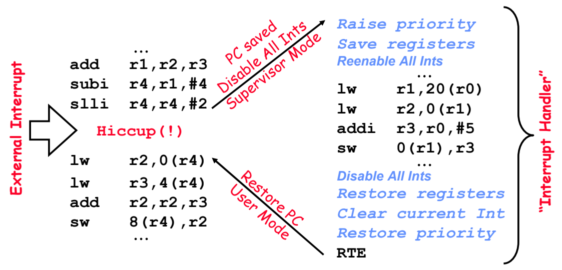
http://www.eecs.berkeley.edu/~culler
http://www-inst.eecs.berkeley.edu/~cs252

Exception

- Unprogrammed change of control flow

Example 1: Device Interrupt

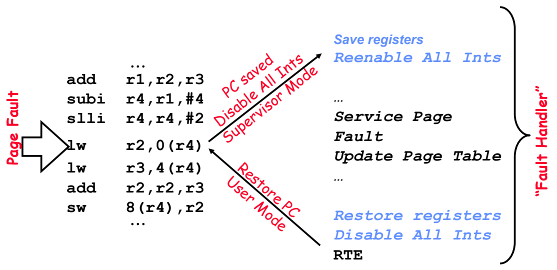
(Say, arrival of network message)



```
         …
add    r1,r2,r3
subi   r4,r1,#4
slli   r4,r4,#2
   Hiccup(!)
lw     r2,0(r4)
lw     r3,4(r4)
add    r2,r2,r3
sw     8(r4),r2
         …
```

```
Raise priority
Save registers
Reenable All Ints
         …
lw     r1,20(r0)
lw     r2,0(r1)
addi   r3,r0,#5
sw     0(r1),r3
         …
Disable All Ints
Restore registers
Clear current Int
Restore priority
RTE
```

Example 2: Page Fault



```
         …
add    r1,r2,r3
subi   r4,r1,#4
slli   r4,r4,#2
lw     r2,0(r4)
lw     r3,4(r4)
add    r2,r2,r3
sw     8(r4),r2
         …
```

```
Save registers
Reenable All Ints
…
Service Page
Fault
Update Page Table
…
Restore registers
Disable All Ints
RTE
```

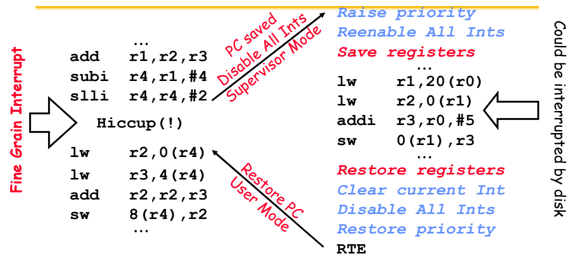
Exception classifications

- *Traps:* relevant to the current process
  - Faults, arithmetic traps, and system "calls"
  - Invoke software on behalf of the currently executing process
- *Interrupts:* caused by asynchronous, outside events
  - I/O devices requiring service (DISK, network)
  - Clock interrupts (real time scheduling)
- *Machine Checks:* caused by serious hardware failure
  - Not always restartable
  - Indicate that bad things have happened.
    - Non-recoverable ECC error
    - Machine room fire
    - Power outage

A related classification: Synchronous vs. Asynchronous

- *Synchronous:* means related to the instruction stream, i.e. during the execution of an instruction
  - Must stop an instruction that is currently executing
  - Page fault on load or store instruction
  - Arithmetic exception
  - Software Trap Instructions
- *Asynchronous:* means unrelated to the instruction stream, i.e. caused by an outside event.
  - Does not have to disrupt instructions that are already executing
  - Interrupts are asynchronous
  - Machine checks are asynchronous
- *SemiSynchronous (or high-availability interrupts):*
  - Caused by external event but may have to disrupt current instructions in order to guarantee service

Can we have fast interrupts?



```
        ...
add     r1,r2,r3
subi    r4,r1,#4
slli    r4,r4,#2
   Hiccup(!)
lw      r2,0(r4)
lw      r3,4(r4)
add     r2,r2,r3
sw      8(r4),r2
        ...
```

```
Raise priority
Reenable All Ints
Save registers
        ...
lw      r1,20(r0)
lw      r2,0(r1)
addi    r3,r0,#5
sw      0(r1),r3
        ...
Restore registers
Clear current Int
Disable All Ints
Restore priority
RTE
```

- Pipeline Drain: Can be very Expensive
- Priority Manipulations
- Register Save/Restore
  – 128 registers + cache misses + etc.

SPARC (and RISC I) had register windows

- On interrupt or procedure call, simply switch to a different set of registers
- Really saves on interrupt overhead
  – Interrupts can happen at any point in the execution, so compiler cannot help with knowledge of live registers.
  – Conservative handlers must save all registers
  – Short handlers might be able to save only a few, but this analysis is compilcated
- Not as big a deal with procedure calls
  – Original statement by Patterson was that Berkeley didn't have a compiler team, so they used a hardware solution
  – Good compilers can allocate registers across procedure boundaries
  – Good compilers know what registers are live at any one time
- However, register windows have returned!
  – IA64 has them
  – Many other processors have shadow registers for interrupts

Supervisor State

- Typically, processors have some amount of state that user programs are not allowed to touch.
  – Page mapping hardware/TLB
    » TLB prevents one user from accessing memory of another
    » TLB protection prevents user from modifying mappings
  – Interrupt controllers -- User code prevented from crashing machine by disabling interrupts. Ignoring device interrupts, etc.
  – Real-time clock interrupts ensure that users cannot lockup/crash machine even if they run code that goes into a loop:
    » "Preemptive Multitasking" vs "non-preemptive multitasking"
- Access to hardware devices restricted
  – Prevents malicious user from stealing network packets
  – Prevents user from writing over disk blocks
- Distinction made with at least two-levels: USER/SYSTEM (one hardware mode-bit)
  – x86 architectures actually provide 4 different levels, only two usually used by OS (or only 1 in older Microsoft OSs)

Entry into Supervisor Mode

- Entry into supervisor mode typically happens on interrupts, exceptions, and special trap instructions.
- Entry goes through kernel instructions:
  – interrupts, exceptions, and trap instructions change to supervisor mode, then jump (indirectly) through table of instructions in kernel

```
intvec: j       handle_int0
        j       handle_int1
                ...
        j       handle_fp_except0
                ...
        j       handle_trap0
        j       handle_trap1
```

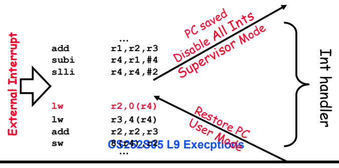
  – OS "System Calls" are just trap instructions:

```
read(fd,buffer,count)  =>   st      20(r0),r1
                            st      24(r0),r2
                            st      28(r0),r3
                            trap    $READ
```

- OS overhead can be serious concern for achieving fast interrupt behavior.

Precise Interrupts/Exceptions

- An interrupt or exception is considered *precise* if there is a single instruction (or interrupt point) for which:
  – All instructions before that have committed their state
  – No following instructions (including the interrupting instruction) have modified any state.
- This means, that you can restart execution at the interrupt point and "get the right answer"
  – Implicit in our previous example of a device interrupt:
    » Interrupt point is at first lw instruction



```
        ...
add     r1,r2,r3
subi    r4,r1,#4
slli    r4,r4,#2

lw      r2,0(r4)
lw      r3,4(r4)
add     r2,r2,r3
sw      8(r4),r2
        ...
```

Precise interrupt point may require multiple PCs

```
       addi  r4,r3,#4
       sub   r1,r2,r3
PC:    bne   r1,there     <=  Interrupt point described as <PC,PC+4>
PC+4:  and   r2,r3,r5
       <other insts>

       addi  r4,r3,#4
       sub   r1,r2,r3          Interrupt point described as:
PC:    bne   r1,there
PC+4:  and   r2,r3,r5     <=   <PC+4,there> (branch was taken)
       <other insts>                    or
                               <PC+4,PC+8> (branch was not taken)
```

- On SPARC, interrupt hardware produces "pc" and "npc" (next pc)
- On MIPS, only "pc" – must fix point in software

Why are precise interrupts desirable?

- Many types of interrupts/exceptions need to be restartable. Easier to figure out what actually happened:
  - I.e. TLB faults. Need to fix translation, then restart load/store
  - IEEE gradual underflow, illegal operation, etc:

    e.g. Suppose you are computing: $f(x)=\frac{\sin(x)}{x}$
    Then, for $x \to 0$

    $$f(0)=\frac{0}{0} \Rightarrow NaN + illegal\_operation$$

    Want to take exception, replace *NaN* with 1, then restart.
- Restartability doesn't *require* preciseness. However, preciseness makes it *a lot easier* to restart.
- Simplify the task of the operating system a lot
  - Less state needs to be saved away if unloading process.
  - Quick to restart (making for fast interrupts)

Approximations to precise interrupts

- Hardware has imprecise state at time of interrupt
- Exception handler must figure out how to find a precise PC at which to restart program.
  - Emulate instructions that may remain in pipeline
  - Example: SPARC allows limited parallelism between FP and integer core:
    - » possible that integer instructions #1 - #4 have already executed at time that the first floating instruction gets a recoverable exception
    - » Interrupt handler code must fixup <float 1>, then emulate both <float 1> and <float 2>
    - » At that point, precise interrupt point is integer instruction #5.

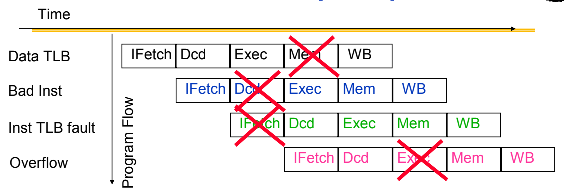
```
<float 1>
<int 1>
<int 2>
<int 3>
<float 2>
<int 4>
<int 5>
```

- Vax had string move instructions that could be in middle at time that page-fault occurred.
- Could be arbitrary processor state that needs to be restored to restart execution.

Precise Exceptions in simple 5-stage pipeline:

- Exceptions may occur at different stages in pipeline (I.e. out of order):
  - Arithmetic exceptions occur in execution stage
  - TLB faults can occur in instruction fetch or memory stage
- What about interrupts? The doctor's mandate of "do no harm" applies here: try to interrupt the pipeline as little as possible
- All of this solved by tagging instructions in pipeline as "cause exception or not" and wait until end of memory stage to flag exception
  - Interrupts become marked NOPs (like bubbles) that are placed into pipeline instead of an instruction.
  - Assume that interrupt condition persists in case NOP flushed
  - Clever instruction fetch might start fetching instructions from interrupt vector, but this is complicated by need for supervisor mode switch, saving of one or more PCs, etc

Another look at the exception problem

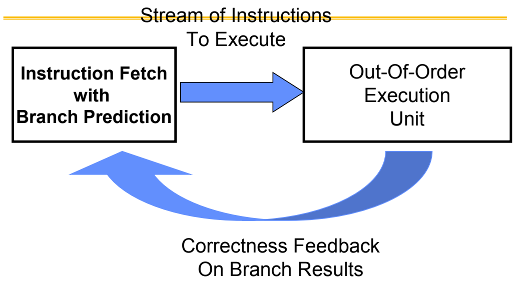


- Use pipeline to sort this out!
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don't act on exception until it reache WB stage
- Handle interrupts through "faulting noop" in IF stage
- When instruction reaches WB stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

How to achieve precise interrupts when instructions executing in arbitrary order?

- Jim Smith's classic paper discusses several methods for getting precise interrupts:
  - In-order instruction completion
  - Reorder buffer
  - History buffer
  - Future buffer

Problem: "Fetch" unit



- Instruction fetch decoupled from execution
- Often issue logic (+ rename) included with Fetch

Branches must be resolved quickly for loop overlap!

- In our loop-unrolling example, we relied on the fact that branches were under control of "fast" integer unit in order to get overlap!

```
Loop:     LD          F0    0     R1
          MULTD       F4    F0    F2
          SD          F4    0     R1
          SUBI        R1    R1    #8
          BNEZ        R1    Loop
```

- What happens if branch depends on result of multd??
  - We completely lose all of our advantages!
  - Need to be able to "predict" branch outcome.
  - If we were to predict that branch was taken, this would be right most of the time.
- Problem much worse for superscalar machines!

Prediction: Branches, Dependencies, Data

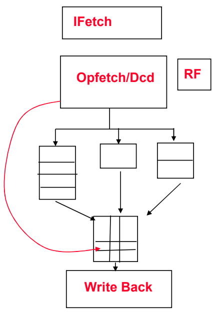
- Prediction has become essential to getting good performance from scalar instruction streams.
- We will discuss predicting branches. However, architects are now predicting everything: *data dependencies, actual data,* and *results of groups of instructions*:
  - At what point does computation become a probabilistic operation + verification?
  - We are pretty close with control hazards already…
- Why does prediction work?
  - Underlying algorithm has regularities.
  - Data that is being operated on has regularities.
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems.
- Prediction ⇒ Compressible information streams?

What about Precise Exceptions/Interrupts?

- Both Scoreboard and Tomasulo have:
  - *In-order issue, out-of-order execution, out-of-order completion*
- Recall: An interrupt or exception is *precise* if there is a single instruction for which:
  - All instructions before that have committed their state
  - No following instructions (including the interrupting instruction) have modified any state.
- Need way to resynchronize execution with instruction stream (I.e. with issue-order)
  - Easiest way is with *in-order completion (i.e. reorder buffer)*
  - Other Techniques (Smith paper): Future File, History Buffer

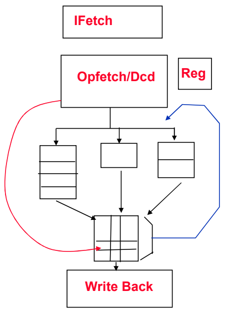
Reorder Buffer

- Idea:
  - record instruction issue order
  - Allow them to execute out of order
  - Reorder them so that they commit in-order
- On issue:
  - Reserve slot at tail of ROB
  - Record dest reg, PC
  - Tag u-op with ROB slot
- Done execute
  - Deposit result in ROB slot
  - Mark exception state
- WB head of ROB
  - Check exception, handle
  - Write register value, or
  - Commit the store



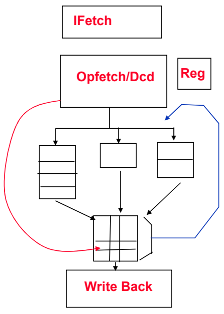
Reorder Buffer + Forwarding

- Idea:
  - Forward uncommitted results to later uncommitted operations
- Trap
  - Discard remainder of ROB
- Opfetch / Exec
  - Match source reg against all dest regs in ROB
  - Forward last (once available)



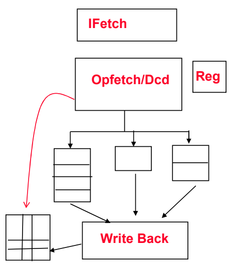
Reorder Buffer + Forwarding + Speculation

- Idea:
  - Issue branch into ROB
  - Mark with prediction
  - Fetch and issue predicted instructions speculatively
  - Branch must resolve before leaving ROB
  - Resolve correct
    - » Commit following instr
  - Resolve incorrect
    - » Mark following instr in ROB as invalid
    - » Let them clear



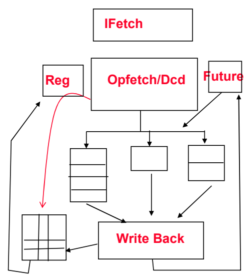
History File

- Maintain issue order, like ROB
- Each entry records dest reg and old value of dest. Register
  - What if old value not available when instruction issues?
- FUs write results into register file
  - Forward into correct entry in history file
- When exception reaches head
  - Restore architected registers from tail to head



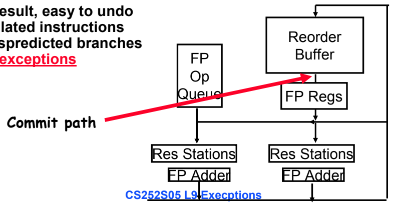
Future file

- Idea
  - Arch registers reflect state at commit point
  - Future register reflect whatever instructions have completed
  - On WB update future
  - On commit update arch
  - On exception
    - » Discard future
    - » Replace with arch
      - Dest w/I ROB



HW support for precise interrupts

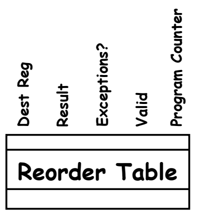
- Concept of Reorder Buffer (ROB):
  - Holds instructions in FIFO order, exactly as they were issued
    - » Each ROB entry contains PC, dest reg, result, exception status
  - When instructions complete, results placed into ROB
    - » Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS
    - » Tag results with ROB buffer number instead of reservation station
  - Instructions commit ⇒values at head of ROB placed in registers
  - As a result, easy to undo speculated instructions on mispredicted branches or on exceptions

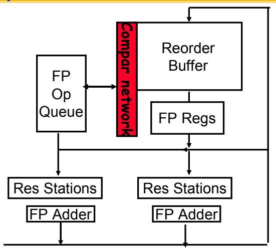


Recall: Four Steps of Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")
2. Execution—operate on operands (EX)
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")
3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
4. Commit—update register with reorder result
   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

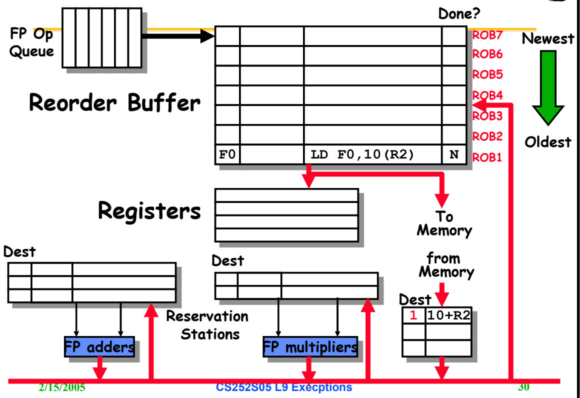
What are the hardware complexities with reorder buffer (ROB)?

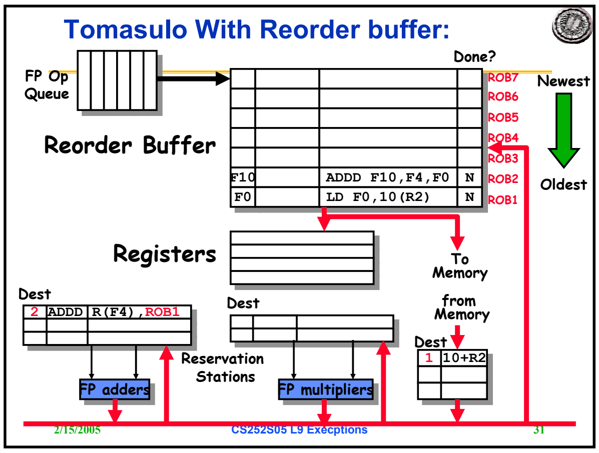




- How do you find the latest version of a register?
  - As specified by Smith paper, need associative comparison network
  - Could use future file or just use the register result status buffer to track which specific reorder buffer has received the value
- Need as many ports on ROB as register file

Tomasulo With Reorder buffer:





Tomasulo With Reorder buffer:

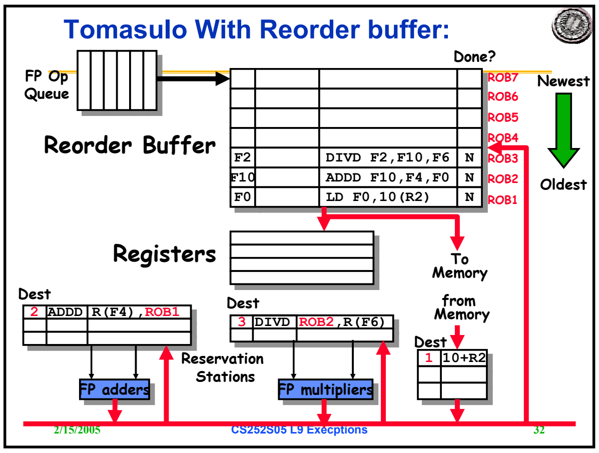
			Done?	
				ROB7
				ROB6
				ROB5
				ROB4
				ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

FP adders reservation stations (Dest):

2	ADDD	R(F4),ROB1

Load buffer (Dest):

1	10+R2



Tomasulo With Reorder buffer:

			Done?	
				ROB7
				ROB6
				ROB5
				ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

FP adders reservation stations (Dest):

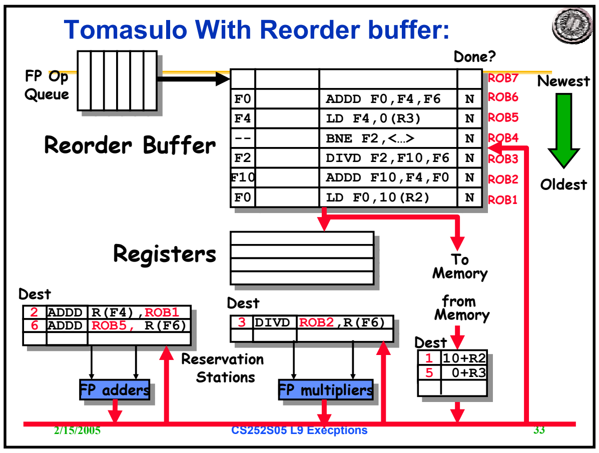
2	ADDD	R(F4),ROB1

FP multipliers reservation stations (Dest):

3	DIVD	ROB2,R(F6)

Load buffer (Dest):

1	10+R2



Tomasulo With Reorder buffer:

			Done?	
				ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4		LD F4,0(R3)	N	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

FP adders reservation stations (Dest):

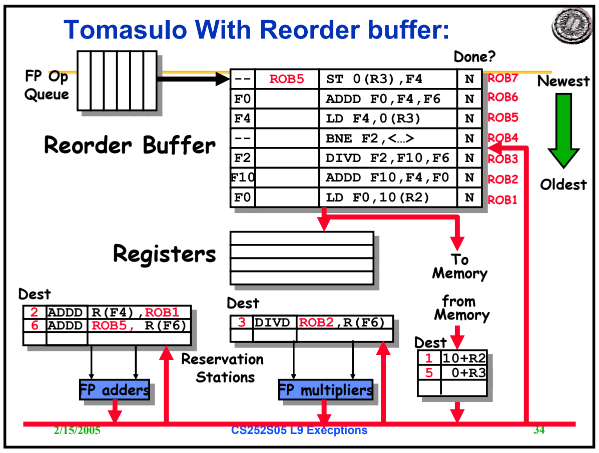
2	ADDD	R(F4),ROB1
6	ADDD	ROB5, R(F6)

FP multipliers reservation stations (Dest):

3	DIVD	ROB2,R(F6)

Load buffer (Dest):

1	10+R2
5	0+R3



Tomasulo With Reorder buffer:

			Done?	
--	ROB5	ST 0(R3),F4	N	ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4		LD F4,0(R3)	N	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

FP adders reservation stations (Dest):

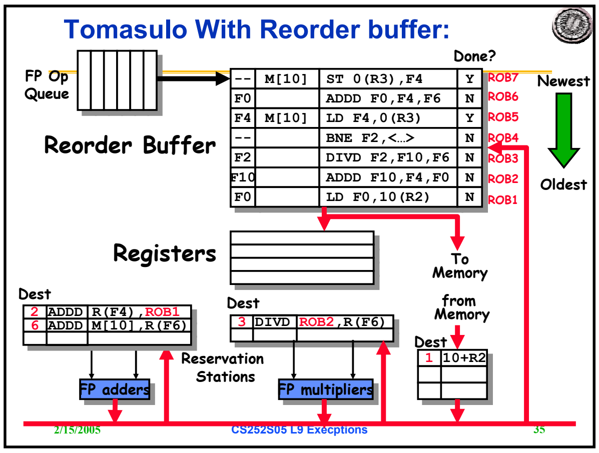
2	ADDD	R(F4),ROB1
6	ADDD	ROB5, R(F6)

FP multipliers reservation stations (Dest):

3	DIVD	ROB2,R(F6)

Load buffer (Dest):

1	10+R2
5	0+R3



Tomasulo With Reorder buffer:

			Done?	
--	M[10]	ST 0(R3),F4	Y	ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4	M[10]	LD F4,0(R3)	Y	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

FP adders reservation stations (Dest):

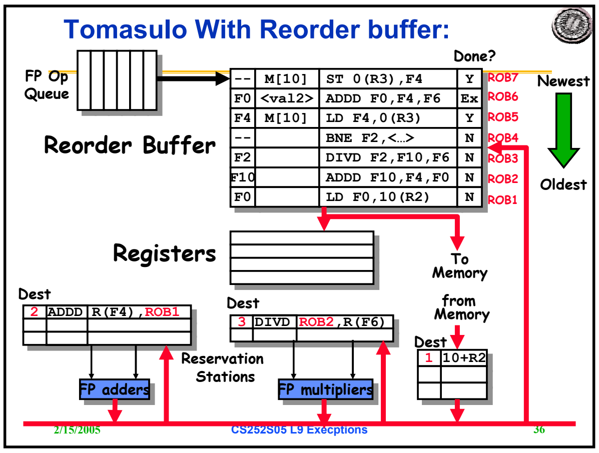
2	ADDD	R(F4),ROB1
6	ADDD	M[10],R(F6)

FP multipliers reservation stations (Dest):

3	DIVD	ROB2,R(F6)

Load buffer (Dest):

1	10+R2



Tomasulo With Reorder buffer:

			Done?	
--	M[10]	ST 0(R3),F4	Y	ROB7
F0	<val2>	ADDD F0,F4,F6	Ex	ROB6
F4	M[10]	LD F4,0(R3)	Y	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

FP adders reservation stations (Dest):

2	ADDD	R(F4),ROB1

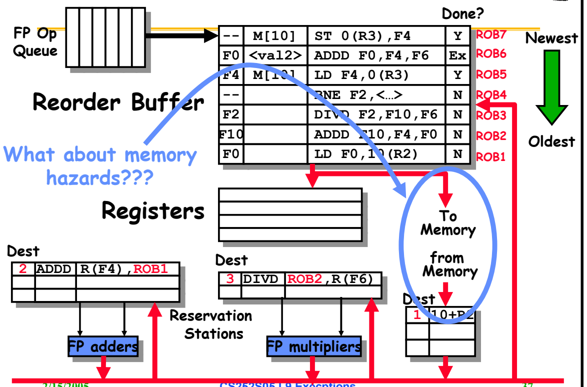
FP multipliers reservation stations (Dest):

3	DIVD	ROB2,R(F6)

Load buffer (Dest):

1	10+R2

Tomasulo With Reorder buffer:



FP Op Queue
Reorder Buffer
Done?
-- | M[10] | ST 0(R3),F4 | Y | ROB7
F0 | <val2> | ADDD F0,F4,F6 | Ex | ROB6
F4 | M[10] | LD F4,0(R3) | Y | ROB5
-- | | BNE F2,<…> | N | ROB4
F2 | | DIVD F2,F10,F6 | N | ROB3
F10 | | ADDD F10,F4,F0 | N | ROB2
F0 | | LD F0,10(R2) | N | ROB1
Newest
Oldest
What about memory hazards???
Registers
Dest
2 ADDD R(F4),ROB1
Dest
3 DIVD ROB2,R(F6)
To Memory
from Memory
Dest
1 10+R2
Reservation Stations
FP adders
FP multipliers

Memory Disambiguation: Sorting out RAW Hazards in memory

- Question: Given a load that follows a store in program order, are the two related?
  - (Alternatively: is there a RAW hazard between the store and the load)?

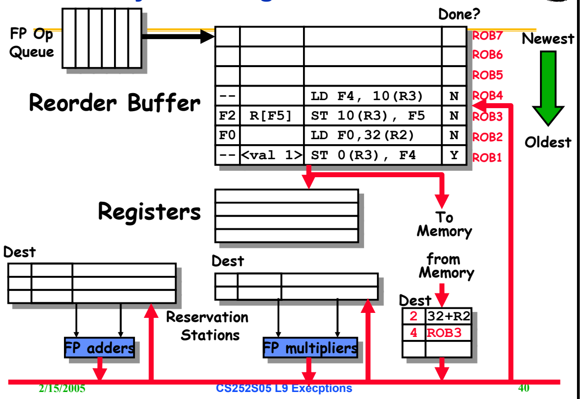
```
Eg:     st      0(R2),R5
        ld      R6,0(R3)
```

- Can we go ahead and start the load early?
  - Store address could be delayed for a long time by some calculation that leads to R2 (divide?).
  - We might want to issue/begin execution of both operations in same cycle.
  - Today: Answer is that we are not allowed to start load until we know that address 0(R2) $\neq$ 0(R3)
  - Later: We might guess at whether or not they are dependent (called "dependence speculation") and use reorder buffer to fixup if we are wrong.

Hardware Support for Memory Disambiguation

- Need buffer to keep track of all outstanding stores to memory, in program order.
  - Keep track of address (when becomes available) and value (when becomes available)
  - FIFO ordering: will retire stores from this buffer in program order
- When issuing a load, record current head of store queue (know which stores are ahead of you).
- When have address for load, check store queue:
  - If *any* store prior to load is waiting for its address, stall load.
  - If load address matches earlier store address (associative lookup), then we have a *memory-induced RAW hazard*:
    - » store value available $\Rightarrow$ return value
    - » store value not available $\Rightarrow$ return ROB number of source
  - Otherwise, send out request to memory
- Actual stores commit in order, so no worry about WAR/WAW hazards through memory.

Memory Disambiguation:

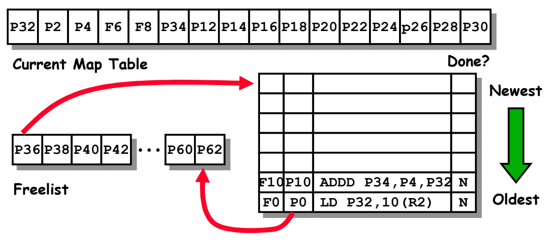


Relationship between precise interrupts and speculation:

- Speculation is a form of guessing
  - Branch prediction, data prediction
  - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
  - This is exactly same as precise exceptions!
- Branch prediction is a very important!
  - Need to "take our best shot" at predicting branch direction.
  - If we issue multiple instructions per cycle, lose lots of potential instructions otherwise:
    - » Consider 4 instructions per cycle
    - » If take single cycle to decide on branch, waste from 4 - 7 instruction slots!
- Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*
  - This is why reorder buffers in all new processors

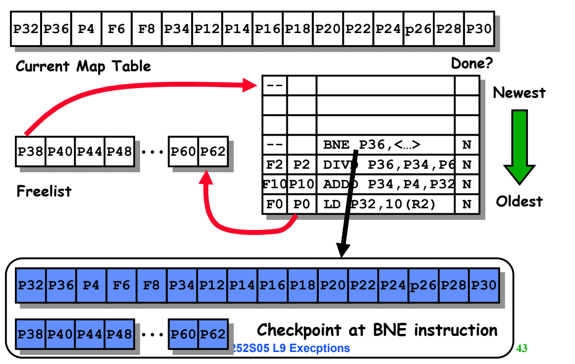
Explicit register renaming:

R10000 Freelist Management

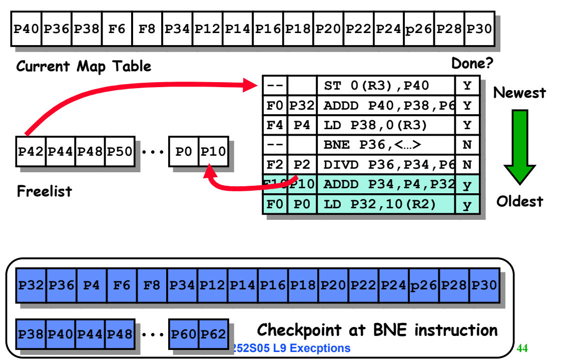


P32 | P2 | P4 | F6 | F8 | P34 | P12 | P14 | P16 | P18 | P20 | P22 | P24 | p26 | P28 | P30
Current Map Table
Done?
F10 | P10 | ADDD P34,P4,P32 | N
F0 | P0 | LD P32,10(R2) | N
Newest
Oldest
P36 | P38 | P40 | P42 | • • • | P60 | P62
Freelist

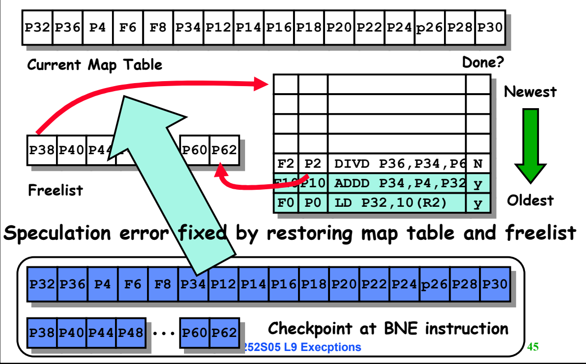
Explicit register renaming:
R10000 Freelist Management


P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
Current Map Table
Done?
Newest
Oldest
-- | | |
-- | | BNE P36,<…> | N
F2 | P2 | DIVD P36,P34,P6 | N
F10 | P10 | ADDD P34,P4,P32 | N
F0 | P0 | LD P32,10(R2) | N
P38 P40 P44 P48 ••• P60 P62
Freelist
P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
P38 P40 P44 P48 ••• P60 P62
Checkpoint at BNE instruction

Explicit register renaming:
R10000 Freelist Management


P40 P36 P38 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
Current Map Table
Done?
Newest
Oldest
-- | | ST 0(R3),P40 | Y
F0 | P32 | ADDD P40,P38,P6 | Y
F4 | P4 | LD P38,0(R3) | Y
-- | | BNE P36,<…> | N
F2 | P2 | DIVD P36,P34,P6 | N
F10 | P10 | ADDD P34,P4,P32 | y
F0 | P0 | LD P32,10(R2) | y
P42 P44 P48 P50 ••• P0 P10
Freelist
P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
P38 P40 P44 P48 ••• P60 P62
Checkpoint at BNE instruction

Explicit register renaming:
R10000 Freelist Management


P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
Current Map Table
Done?
Newest
Oldest
F2 | P2 | DIVD P36,P34,P6 | N
F10 | P10 | ADDD P34,P4,P32 | y
F0 | P0 | LD P32,10(R2) | y
P38 P40 P44 P60 P62
Freelist
P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
P38 P40 P44 P48 ••• P60 P62
Checkpoint at BNE instruction

Speculation error fixed by restoring map table and freelist

Summary

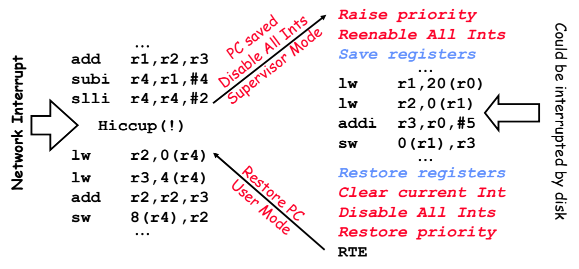
- Control flow causes lots of trouble with pipelining
  - Other hazards can be "fixed" with more transistors or forwarding
  - We will spend a lot of time on branch prediction techniques
- Some pre-decode techniques can transform dynamic decisions into static ones (VLIW-like)
  - Beginnings of dynamic compilation techniques
- Interrupts and Exceptions either interrupt the current instruction or happen between instructions
  - Possibly large quantities of state must be saved before interrupting
- Machines with *precise exceptions* provide one single point in the program to restart execution
  - All instructions before that point have completed
  - No instructions after or including that point have completed
- Hardware techniques exist for precise exceptions even in the face of out-of-order execution!
  - Important enabling factor for out-of-order execution

Alternative: Polling
(again, for arrival of network message)

External Interrupt

```
          Disable Network Intr
                   …
          subi     r4,r1,#4
          slli     r4,r4,#2
          lw       r2,0(r4)
          lw       r3,4(r4)
          add      r2,r2,r3
          sw       8(r4),r2
          lw       r1,12(r0)        } Polling Point
          beq      r1,no_mess       } (check device register)
          lw       r1,20(r0)        }
          lw       r2,0(r1)         }
          addi     r3,r0,#5         }  "Handler"
          sw       0(r1),r3         }
          Clear Network Intr        }
no_mess:           …
```

Interrupt Priorities Must be Handled



```
             …                   Raise priority
add      r1,r2,r3                Reenable All Ints
subi     r4,r1,#4                Save registers
slli     r4,r4,#2                       …
                                 lw      r1,20(r0)
  Hiccup(!)                      lw      r2,0(r1)
                                 addi    r3,r0,#5
lw       r2,0(r4)                sw      0(r1),r3
lw       r3,4(r4)                       …
add      r2,r2,r3                Restore registers
sw       8(r4),r2                Clear current Int
             …                   Disable All Ints
                                 Restore priority
                                 RTE
```

Note that priority must be raised to avoid recursive interrupts!

Interrupt controller hardware and mask levels

- Operating system constructs a hierarchy of masks that reflects some form of interrupt priority.
- For instance:

Priority	Examples
0	Software interrupts
2	Network Interrupts
4	Sound card
5	Disk Interrupt
6	Real Time clock
🕒	Non-Maskable Ints (power)

  – This reflects the an order of urgency to interrupts
  – For instance, this ordering says that disk events can interrupt the interrupt handlers for network interrupts.

Polling is faster/slower than Interrupts.

- Polling is faster than interrupts because
  – Compiler knows which registers in use at polling point. Hence, do not need to save and restore registers (or not as many).
  – Other interrupt overhead avoided (pipeline flush, trap priorities, etc).
- Polling is slower than interrupts because
  – Overhead of polling instructions is incurred regardless of whether or not handler is run. This could add to inner-loop delay.
  – Device may have to wait for service for a long time.
- When to use one or the other?
  – Multi-axis tradeoff
    » Frequent/regular events good for polling, *as long as device can be controlled at user level.*
    » Interrupts good for infrequent/irregular events
    » Interrupts good for ensuring regular/predictable service of events.