EECS 252 Graduate Computer Architecture

Lec 5 – Out-of-Order Completion

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

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

Review

- Data stationary pipeline control
  - Micro-instruction & PC track down the pipe
  - Accumulate state
- Implementing bubbles, stalls, forwarding, multicycle operations
- Branch prediction
  - Static vs dynamic
  - N-bit saturating counters
  - Local and global history
  - Correlated predictors, Tournament, GSHARE
  - Branch target buffers, return address predictors

Outline

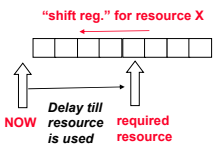
- Relax pipeline design to allow out-of-order completions
  - Cray-1: register reservations
- Relax pipeline to allow out-of-order issue
  - CDC 6600: Scoreboard
- Compiler optimizations for ILP
- Superscalar issue
- Maybe Go back and finish exceptions

Pipelining with Reg. Reservations

- Assumptions
  1. Multiple pipelined function units of different latency
     - » able to accept operations at issue rate
     - » may be exceptions (e.g., divide)
  2. Issue instructions in order
  3. Operand fetch in order
  4. Completion out of order
     - » short ops may bypass long ones
  5. Some shared resources (e.g., reg write port)
- Implications
  - WAR hazard still resolved by pipeline flow (2 & 3)
  - RAW, WAW, and structural still present
- Design philosophy (ala Cray)
  - Resolve hazards as instruction is issued into pipeline
  - Pipeline is non-blocking

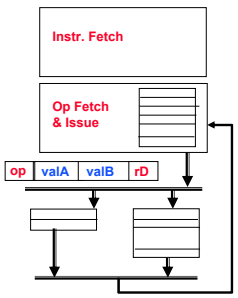
Resolving Structural Hazards

- With static pipeline flow, resource usage is known in advance
- Instruction requires X at *t* ticks after issue
- If $reservation_X[t]$ is clear, issue inst and set bit
- Otherwise, delay till clear
- At each tick the $reservation_X[]$ shifts by one, so will eventually clear
- Multiple resources? Range of delays?



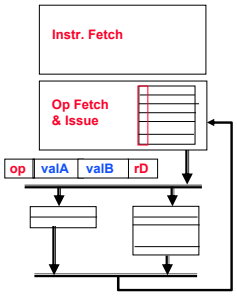
Basic Issue Model

- Issue unit checks for all hazards
  - Structural RAW, WAW
- Holds issue while hazards exist
- Upon issue, register values provided to F.U
- Executes to completion without blocking



Hazard Resolution

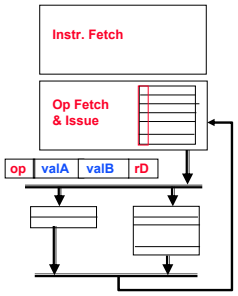
- Structural
  - Op code => resource usage
  - Check resource resv
  - Set on issue
- Data
  - Add reservation bit one each register
  - Check RegRsv for source and destination registers
  - Hold issue till clear
  - Set bit on destination register
  - Clear bit on dest reg. Write
- Questions:
  - Forwarding?



Motorola 88000 "scoreboard" [sic]

Example

```
Add r1 := r2 + r3
Add r2 := r2 + 4
Lod r5 := mem[r1+16]
Lod r6 := mem[r1+32]
Mul r7 := r5 * r6
Bnz r1, foo
Sub  r7 := r0 - r0
```



Cray-1 Discussion

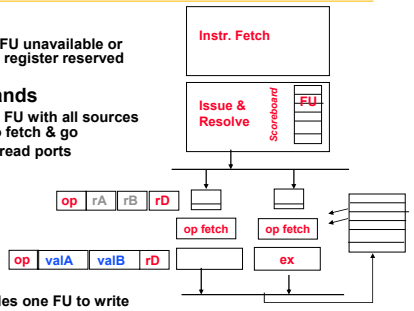
- Technological Assumptions
- Why no forwarding?
- Longevity of the ISA?
- Instruction cache?
  - Four blocks (RR) of 16x4 "parcels"
  - Issue delayed on miss
    - » 2 CP for change of block
- Branch delays?
  - Brach op code delayed till second parcel is obtained
  - 5 clocks (reg zero, nz, pos, neg)
- I/O system?

Pipelining with Scoreboarding

- Assumptions
  1. Multiple function units of different latency
     - Especially non-pipelined units
  2. Issue instructions whenever FU available, unless would cause multiple outstanding writes to same regsiter
     - Operand fetch out of order
     - Completion out of order
  3. Some shared resources (e.g., reg write port)
- Implications
  - Need to resolve RAW, WAR, WAW and structural
- Design philosophy (ala CDC 6600)
  - Issue unit tracks all outstanding dependences
  - Holds issue if structural or WAW hazard
  - Informs FUs when hazards resolved
  - FUs fetch operands from register file and proceed

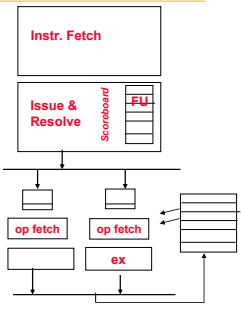
Scoreboard Operation

- Issue
  - Hold while FU unavailable or destination register reserved (by FU *f*)
- Read operands
  - SB informs FU with all sources available to fetch & go
  - Limited by read ports
- Write back
  - SB schedules one FU to write
  - Waits no FU waiting to fetch (old version) of reg



Example

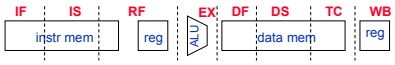
```
Add r1 := r2 + r3
Add r2 := r2 + 4
Lod r5 := mem[r1+16]
Lod r6 := mem[r1+32]
Mul r7 := r5 * r6
Bnz r1, foo
Sub  r7 := r0 - r0
```



Discussion

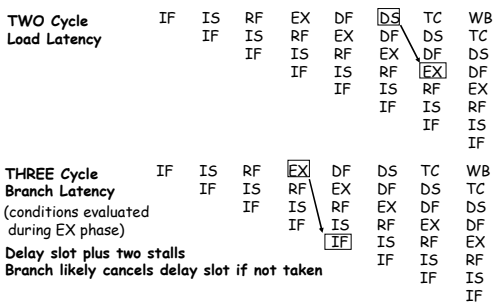
- Technological Assumptions
- Extend to allow forwarding?
- How do loads and stores work?
- Instruction cache?
- I/O system?

Case Study: MIPS R4000 (200 MHz)



- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- 8 Stages: What is impact on Load delay? Branch delay? Why?

Case Study: MIPS R4000



MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Last step of FP Multiplier/Divider uses FP Adder HW
- 8 kinds of stages in FP units:

Stage	*Functional unit*	*Description*
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

MIPS FP Pipe Stages

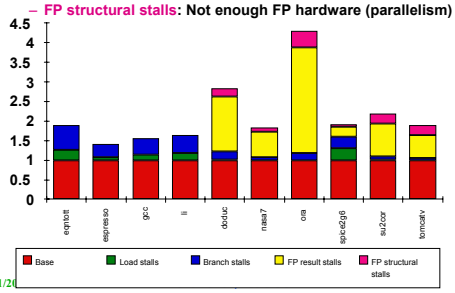
FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	…
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D^{28}	…	D+A	D+R, D+R, D+A, D+R, A, R		
Square root	U	E	$(A+R)^{108}$	…	A	R			
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

Stages:

M	*First stage of multiplier*	*A*	*Mantissa ADD stage*
N	*Second stage of multiplier*	*D*	*Divide pipeline stage*
R	*Rounding stage*	*E*	*Exception test stage*
S	*Operand shift stage*		
U	*Unpack FP numbers*		

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)



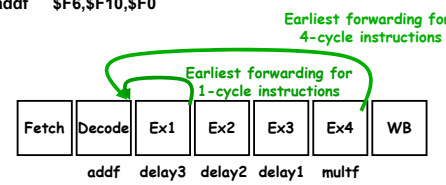
Advanced Pipelining and Instruction Level Parallelism (ILP)

- ILP: Overlap execution of unrelated instructions
- gcc 17% control transfer
  - 5 instructions + 1 branch
  - Beyond single block to get more instruction level parallelism
- Loop level parallelism one opportunity
  - First SW, then HW approaches
- DLX Floating Point as example
  - Measurements suggests R4000 performance FP execution has room for improvement

Can we make CPI closer to 1?

- Let's assume full pipelining:
  - If we have a 4-cycle *latency*, then we need 3 instructions between a producing instruction and its use:

```
multf   $F0,$F2,$F4
delay-1
delay-2
delay-3
addf    $F6,$F10,$F0
```



FP Loop: Where are the Hazards?

```
Loop:  LD    F0,0(R1)   ;F0=vector element
       ADDD  F4,F0,F2   ;add scalar from F2
       SD    0(R1),F4   ;store result
       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
       BNEZ  R1,Loop    ;branch R1!=zero
       NOP              ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- Where are the stalls?

FP Loop Showing Stalls

```
1 Loop: LD    F0,0(R1)   ;F0=vector element
2       stall
3       ADDD  F4,F0,F2   ;add scalar in F2
4       stall
5       stall
6       SD    0(R1),F4   ;store result
7       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
8       BNEZ  R1,Loop    ;branch R1!=zero
9       stall            ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: LD    F0,0(R1)
2       stall
3       ADDD  F4,F0,F2
4       SUBI  R1,R1,8
5       BNEZ  R1,Loop    ;delayed branch
6       SD    8(R1),F4   ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks: Unroll loop 4 times code to make faster?

Unroll Loop Four Times (straightforward way)

```
1 Loop: LD    F0,0(R1)
2       ADDD  F4,F0,F2
3       SD    0(R1),F4    ;drop SUBI & BNEZ
4       LD    F6,-8(R1)
5       ADDD  F8,F6,F2
6       SD    -8(R1),F8   ;drop SUBI & BNEZ
7       LD    F10,-16(R1)
8       ADDD  F12,F10,F2
9       SD    -16(R1),F12 ;drop SUBI & BNEZ
10      LD    F14,-24(R1)
11      ADDD  F16,F14,F2
12      SD    -24(R1),F16
13      SUBI  R1,R1,#32   ;alter to 4*8
14      BNEZ  R1,LOOP
15      NOP
```

1 cycle stall
2 cycles stall

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4

Unrolled Loop That Minimizes Stalls

```
1 Loop:LD    F0,0(R1)
2      LD    F6,-8(R1)
3      LD    F10,-16(R1)
4      LD    F14,-24(R1)
5      ADDD  F4,F0,F2
6      ADDD  F8,F6,F2
7      ADDD  F12,F10,F2
8      ADDD  F16,F14,F2
9      SD    0(R1),F4
10     SD    -8(R1),F8
11     SD    -16(R1),F12
12     SUBI  R1,R1,#32
13     BNEZ  R1,LOOP
14     SD    8(R1),F16    ; 8-32 = -24
```

- What assumptions made when moved code?
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*Pipe Stages*				
Int. instruction	IF	ID	EX	MEM	WB
FP instruction	IF	ID	EX	MEM	WB
Int. instruction	IF	ID	EX	MEM	WB
FP instruction		IF	ID	EX	MEM	WB
Int. instruction		IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can't use it, nor instructions in next slot

SuperScalar Issue Rules

- Datapath has specific kinds of functional parallelism
- Fetch packet of instructions
- "Issue rules": constraints over and beyond dependencies
  - Ex: one arithmetic or branch, one load/store, one FP

Loop Unrolling in Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration (1.5X)

VLIW: Very Large Instruction Word

- Each "instruction" has explicit coding for multiple operations
  - In EPIC, grouping called a "packet"
  - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
  - In 1976 (same year as Cray-1) Floating Point Systems AP120B
    - » "poor mans Cray", 2 MFLOPS for 50k vs 20 MFLOPS for 12M
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - » 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency
Note: Need more registers in VLIW (15 vs. 6 in SS)

Summary

- Increasingly powerful (and complex) dynamic mechanism for detecting and resolving hazards
  - In-order pipeline, in-order op-fetch with register reservations, in-order issue with scoreboard
  - Weaken the timing and flow assumptions
  - Allow later instructions to proceed around ones that are stalled
  - Facilitate multiple issue
  - Not quite powerful enough to unroll loops dynamically
    - » Stop when attempt to rebind a new value to a reg.
- Compiler techniques make it easier for HW to find the ILP
  - Reduces the impact of more sophisticated organization
  - Requires a larger architected namespace
  - Easier for more structured code