

ECE/CS 552: Pipelining

Instructor: Mikko H Lipasti

Fall 2010
University of Wisconsin-Madison

Lecture notes based on set created by Mark Hill
and John P. Shen
Updated by Mikko Lipasti

Pipelining

- Forecast
 - Big Picture
 - Datapath
 - Control
 - Data Hazards
 - Stalls
 - Forwarding
 - Control Hazards
 - Exceptions

Motivation

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) (CPI) (cycle time)

- Single cycle implementation
 - CPI = 1
 - Cycle = imem + RFrd + ALU + dmem + RFwr + muxes + control
 - E.g. 500+250+500+500+250+0+0 = 2000ps
 - Time/program = P x 2ns

Multicycle

- Multicycle implementation:

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instr:										0	1	2	3
i		F	D	X	M	W							
i+1						F	D	X					
i+2									F	D	X	M	
i+3													F
i+4													

Multicycle

- Multicycle implementation
 - CPI = 3, 4, 5
 - Cycle = max(memory, RF, ALU, mux, control)
 - = max(500,250,500) = 500ps
 - Time/prog = P x 4 x 500 = P x 2000ps = P x 2ns
- Would like:
 - CPI = 1 + overhead from hazards (later)
 - Cycle = 500ps + overhead
 - In practice, ~3x improvement

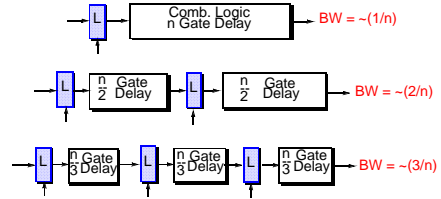
Big Picture

- Instruction latency = 5 cycles
- Instruction throughput = 1/5 instr/cycle
- CPI = 5 cycles per instruction
- Instead
 - Pipelining: process instructions like a lunch buffet
 - ALL microprocessors use it
 - E.g. Core i7, AMD Barcelona, ARM11

Big Picture

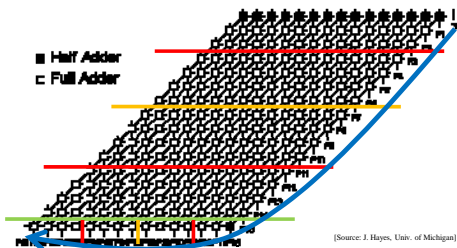
- Instruction Latency = 5 cycles (same)
- Instruction throughput = 1 instr/cycle
- CPI = 1 cycle per instruction
- CPI = cycle between instruction completion = 1

Ideal Pipelining



- Bandwidth increases linearly with pipeline depth
- Latency increases by latch delays

Example: Integer Multiplier



- 16x16 combinational multiplier
 - ISCAS-85 C6288 standard benchmark
 - Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

Example: Integer Multiplier

Configuration	Delay	MPS	Area (FF/wiring)	Area Increase
Combinational	3.52ns	284	7535 (~/1759)	
2 Stages	1.87ns	534 (1.9x)	8725 (1078/1870)	16%
4 Stages	1.17ns	855 (3.0x)	11276 (3388/2112)	50%
8 Stages	0.80ns	1250 (4.4x)	17127 (8938/2612)	127%

- Pipeline efficiency
 - 2-stage: nearly double throughput; marginal area cost
 - 4-stage: 75% efficiency; area still reasonable
 - 8-stage: 55% efficiency; area more than doubles
- Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

Ideal Pipelining

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instr:										0	1	2	3
i	F	D	X	M	W								
i+1		F	D	X	M	W							
i+2			F	D	X	M	W						
i+3				F	D	X	M	W					
i+4					F	D	X	M	W				

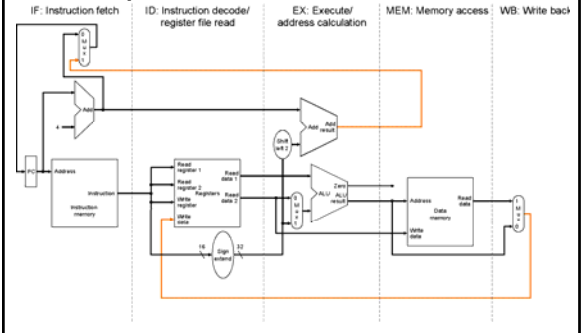
Pipelining Idealisms

- Uniform subcomputations
 - Can pipeline into stages with equal delay
- Identical computations
 - Can fill pipeline with identical work
- Independent computations
 - No relationships between work units
- Are these practical?
 - No, but can get close enough to get significant speedup

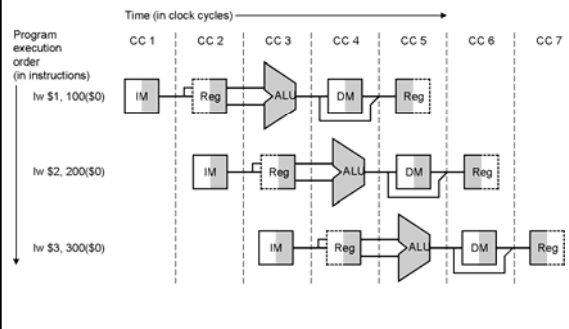
Complications

- Datapath
 - Five (or more) instructions in flight
- Control
 - Must correspond to multiple instructions
- Instructions may have
 - data and control flow *dependences*
 - I.e. units of work are not independent
 - One may have to stall and wait for another

Datapath



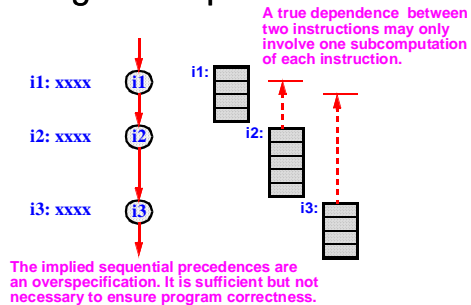
Datapath



Control

- Control
 - Set by 5 different instructions
 - Divide and conquer: carry IR down the pipe
- MIPS ISA requires the appearance of *sequential execution*
 - *Precise exceptions*
 - True of most general purpose ISAs

Program Dependences



Program Data Dependences

- True dependence (RAW) $D(i) \cap R(j) \neq \phi$
 - j cannot execute until i produces its result
- Anti-dependence (WAR) $R(i) \cap D(j) \neq \phi$
 - j cannot write its result until i has read its sources
- Output dependence (WAW) $D(i) \cap D(j) \neq \phi$
 - j cannot write its result until i has written its result

Control Dependences

- Conditional branches
 - Branch must execute to determine which instruction to fetch next
 - Instructions following a conditional branch are control dependent on the branch instruction

Example (quicksort/MIPS)

```

# for ( ; (j < high) && (array[j] < array[low]) ; ++j );
# $t0 = j
# $t9 = high
# $s6 = array
# $s8 = low
done: $t0, $s8
mul   $t5, $t0, 4
addu  $s24, $s6, $t5
lw    $t25, 0($s24)
mul   $t13, $s8, 4
addu  $t14, $s6, $t13
lw    $t15, 0($t14)
bge   done, $t25, $t15

cont:
addu  $t10, $t0, 1
...
done:
addu  $t11, $t11, -1
    
```

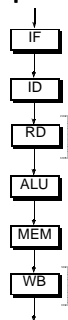
Resolution of Pipeline Hazards

- Pipeline hazards
 - Potential violations of program dependences
 - Must ensure program dependences are not violated
- Hazard resolution
 - Static: compiler/programmer guarantees correctness
 - Dynamic: hardware performs checks at runtime
- Pipeline interlock
 - Hardware mechanism for dynamic hazard resolution
 - Must detect and enforce dependences at runtime

Pipeline Hazards

- Necessary conditions:
 - WAR: write stage earlier than read stage
 - Is this possible in IF-RD-EX-MEM-WB ?
 - WAW: write stage earlier than write stage
 - Is this possible in IF-RD-EX-MEM-WB ?
 - RAW: read stage earlier than write stage
 - Is this possible in IF-RD-EX-MEM-WB ?
- If conditions not met, no need to resolve
- Check for both register and memory

Pipeline Hazard Analysis



- Memory hazards
 - RAW: Yes/No?
 - WAR: Yes/No?
 - WAW: Yes/No?
- Register hazards
 - RAW: Yes/No?
 - WAR: Yes/No?
 - WAW: Yes/No?

RAW Hazard

- Earlier instruction produces a value used by a later instruction:
 - add \$t1, \$t2, \$t3
 - sub \$t4, \$t5, \$t1

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instr:									0	1	2	3	
add	F	D	X	M	W								
sub		F	D	X	M	W							

RAW Hazard - Stall

- Detect dependence and stall:
 - add \$1, \$2, \$3
 - sub \$4, \$5, \$1

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instr:									0	1	2	3	
add	F	D	X	M	W								
sub						F	D	X	M	W			

Control Dependence

- One instruction affects which executes next
 - sw \$4, 0(\$5)
 - bne \$2, \$3, loop
 - sub \$6, \$7, \$8

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instr:										0	1	2	3
sw	F	D	X	M	W								
bne		F	D	X	M	W							
sub			F	D	X	M	W						

Control Dependence - Stall

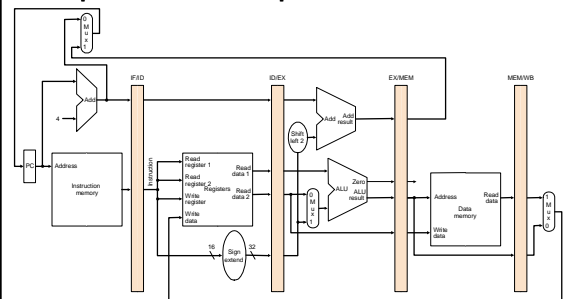
- Detect dependence and stall
 - sw \$4, 0(\$5)
 - bne \$2, \$3, loop
 - sub \$6, \$7, \$8

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instr:									0	1	2	3	
sw	F	D	X	M	W								
bne		F	D	X	M	W							
sub				F	D	X	M	W					

Pipelined Datapath

- Start with single-cycle datapath
- Pipelined execution
 - Assume each instruction has its own datapath
 - But each instruction uses a different part in every cycle
 - Multiplex all on to one datapath
 - Latches separate cycles (like multicycle)
- Ignore hazards for now
 - Data
 - control

Pipelined Datapath



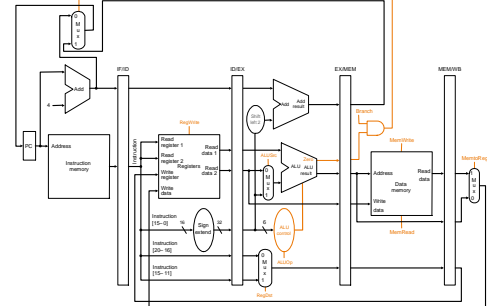
Pipelined Datapath

- Instruction flow
 - add and load
 - Write of registers
 - Pass register specifiers
- Any info needed by a later stage gets passed down the pipeline
 - E.g. store value through EX

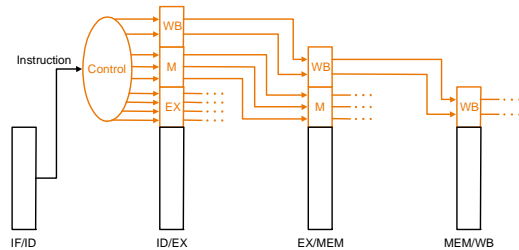
Pipelined Control

- IF and ID
 - None
- EX
 - ALUop, ALUsrc, RegDst
- MEM
 - Branch, MemRead, MemWrite
- WB
 - MemtoReg, RegWrite

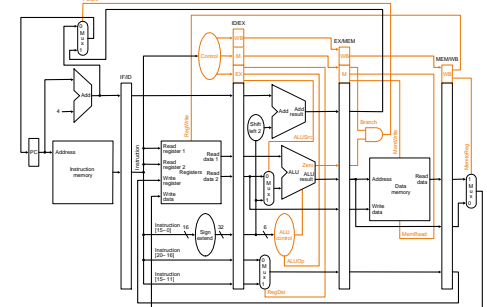
Datapath Control Signals



Pipelined Control



All Together



Pipelined Control

- Controlled by different instructions
- Decode instructions and pass the signals down the pipe
- Control sequencing is embedded in the pipeline
 - No explicit FSM
 - Instead, distributed FSM

Pipelining

- Not too complex yet
 - Data hazards
 - Control hazards
 - Exceptions

RAW Hazards

- Must first detect RAW hazards
 - Pipeline analysis proves that WAR/WAW don't occur
- ID/EX.WriteRegister = IF/ID.ReadRegister1
 ID/EX.WriteRegister = IF/ID.ReadRegister2
 EX/MEM.WriteRegister = IF/ID.ReadRegister1
 EX/MEM.WriteRegister = IF/ID.ReadRegister2
 MEM/WB.WriteRegister = IF/ID.ReadRegister1
 MEM/WB.WriteRegister = IF/ID.ReadRegister2

RAW Hazards

- Not all hazards because
 - WriteRegister not used (e.g. sw)
 - ReadRegister not used (e.g. addi, jump)
 - Do something only if necessary

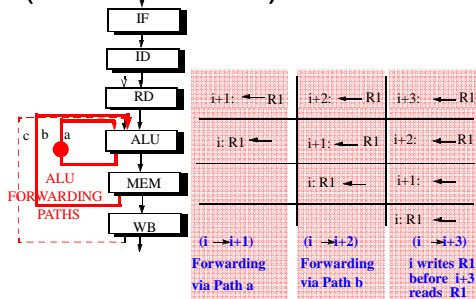
RAW Hazards

- Hazard Detection Unit
 - Several 5-bit (or 6-bit) comparators
- Response? Stall pipeline
 - Instructions in IF and ID stay
 - IF/ID pipeline latch not updated
 - Send 'nop' down pipeline (called a bubble)
 - PCWrite, IF/IDWrite, and nop mux

RAW Hazard Forwarding

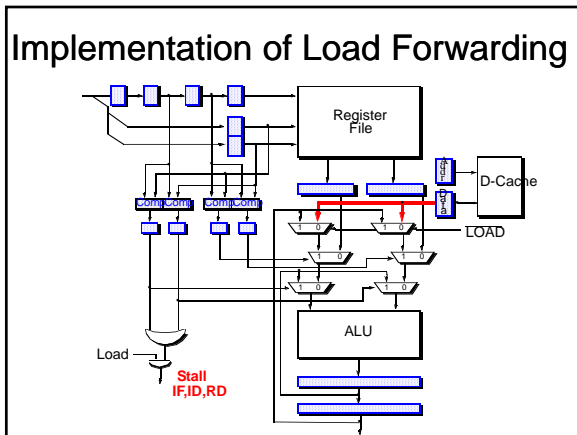
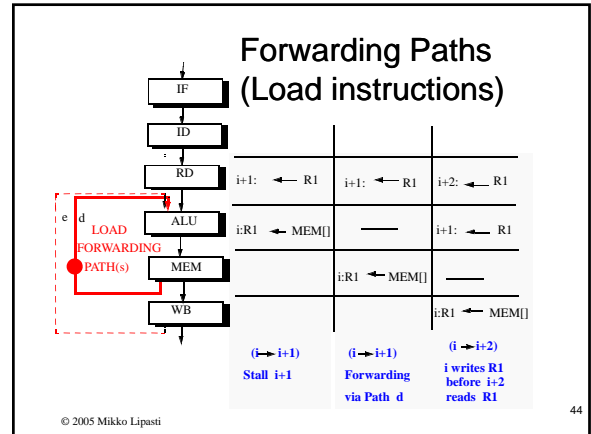
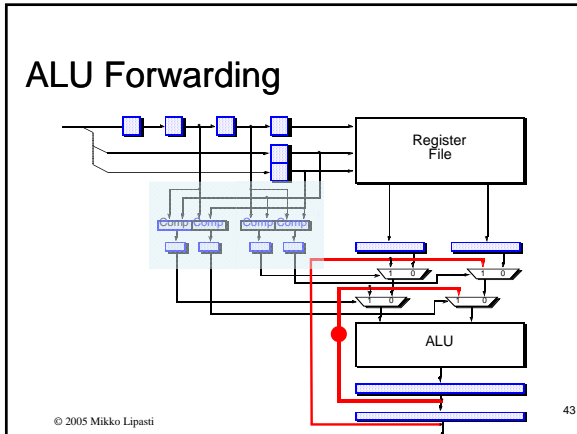
- A better response – forwarding
 - Also called bypassing
- Comparators ensure register is read after it is written
- Instead of stalling until write occurs
 - Use mux to select forwarded value rather than register value
 - Control mux with hazard detection logic

Forwarding Paths (ALU instructions)



Write before Read RF

- Register file design
 - 2-phase clocks common
 - Write RF on first phase
 - Read RF on second phase
- Hence, same cycle:
 - Write \$1
 - Read \$1
- No bypass needed
 - If read before write or DFF-based, need bypass



Control Flow Hazards

- Control flow instructions
 - branches, jumps, jals, returns
 - Can't fetch until branch outcome known
 - Too late for next IF

Control Flow Hazards

- What to do?
 - Always stall
 - Easy to implement
 - Performs poorly
 - 1/6th instructions are branches, each branch takes 3 cycles
 - CPI = 1 + 3 x 1/6 = 1.5 (lower bound)

Control Flow Hazards

- Predict branch not taken
- Send sequential instructions down pipeline
- Kill instructions later if incorrect
- Must stop memory accesses and RF writes
- Late flush of instructions on misprediction
 - Complex
 - Global signal (wire delay)

Control Flow Hazards

- Even better but more complex
 - Predict taken
 - Predict both (eager execution)
 - Predict one or the other dynamically
 - Adapt to program branch patterns
 - Lots of chip real estate these days
 - Pentium III, 4, Alpha 21264
 - Current research topic
 - More later (lecture on branch prediction)

Control Flow Hazards

- Another option: delayed branches
 - Always execute following instruction
 - “delay slot” (later example on MIPS pipeline)
 - Put useful instruction there, otherwise ‘nop’
- A mistake to cement this into ISA
 - Just a stopgap (one cycle, one instruction)
 - Superscalar processors (later)
 - Delay slot just gets in the way (special case)

Exceptions and Pipelining

- add \$1, \$2, \$3 overflows
- A surprise branch
 - Earlier instructions flow to completion
 - Kill later instructions
 - Save PC in EPC, set PC to EX handler, etc.
- Costs a lot of designer sanity
 - 554 teams that try this sometimes fail

Exceptions

- Even worse: in one cycle
 - I/O interrupt
 - User trap to OS (EX)
 - Illegal instruction (ID)
 - Arithmetic overflow
 - Hardware error
 - Etc.
- Interrupt priorities must be supported

Review

- Big Picture
- Datapath
- Control
 - Data hazards
 - Stalls
 - Forwarding or bypassing
 - Control flow hazards
 - Branch prediction
- Exceptions