Pipelined Datapath

Lecture notes from MKP, H. H. Lee and S. Yalamanchili

Reading

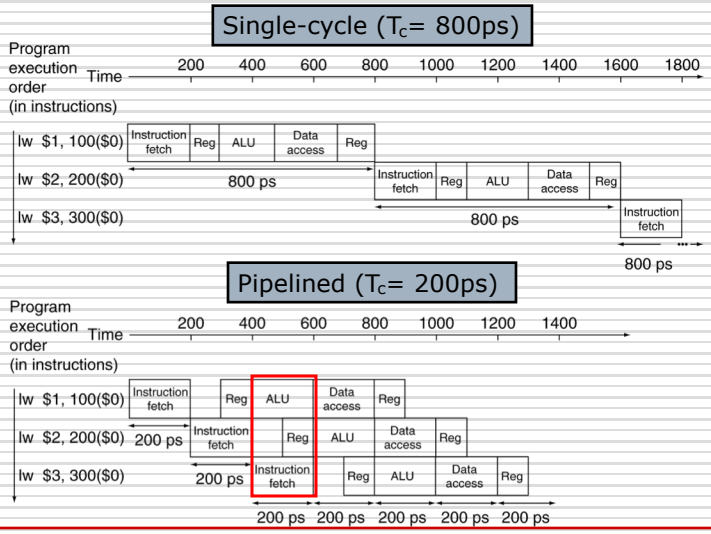
- Sections 4.5 – 4.10
- Practice Problems: 1, 3, 8, 12
- Note: Appendices A-E in the hardcopy text correspond to chapters 7-11 in the online text.

Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time

$$Inter-instruction-gap_{pipelined} = \frac{Inter-instruction-gap_{nonpipelined}}{number-of-stages}$$

- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

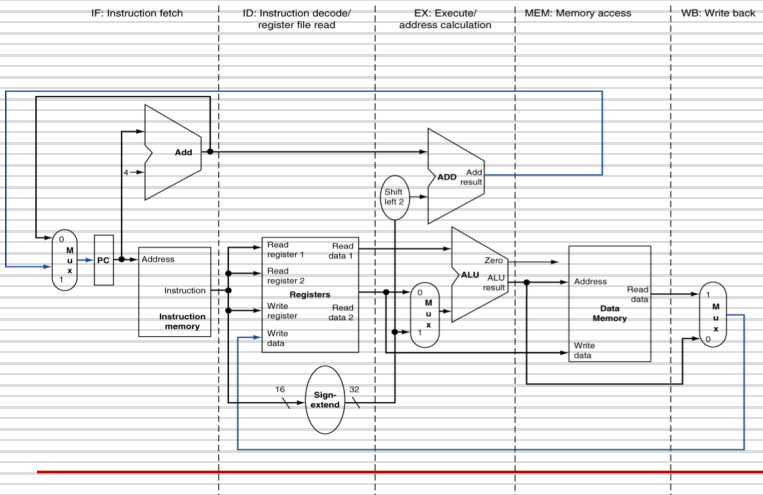
(5)

Basic Idea

All instructions are 32-bits

Few & regular instruction formats

Alignment of memory operands



(6)

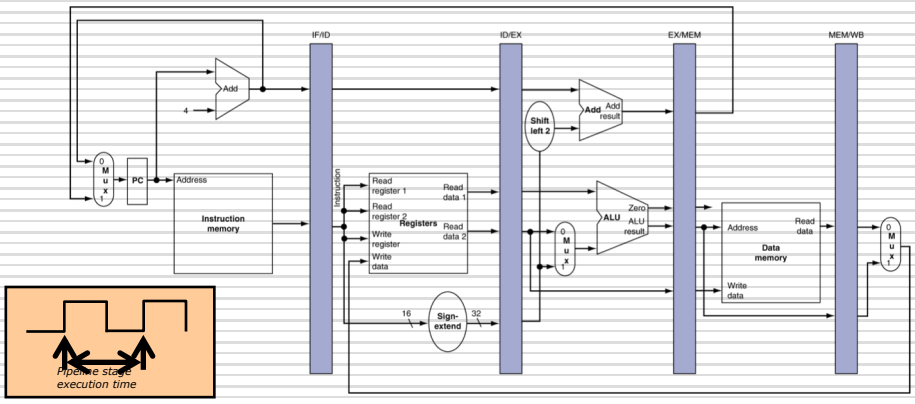
Pipelining

- What makes it easy
  - All instructions are the same length
  - Simple instruction formats
  - Memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction
- What really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

(7)

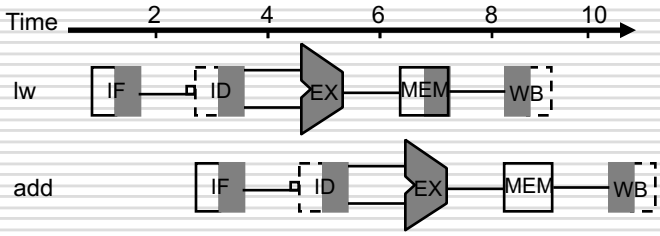
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle



(8)

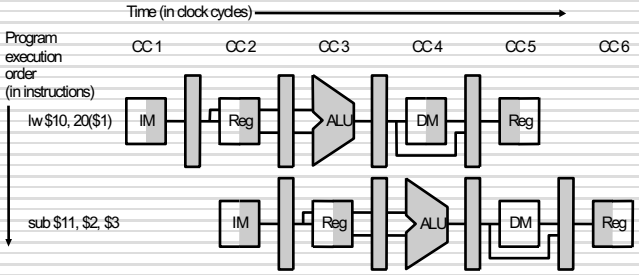
Graphically Representing Pipelines



- Shading indicates the unit is being used by the instruction
- Shading on the right half of the register file (ID or WB) or memory means the element is being read in that stage
- Shading on the left half means the element is being written in that stage

(9)

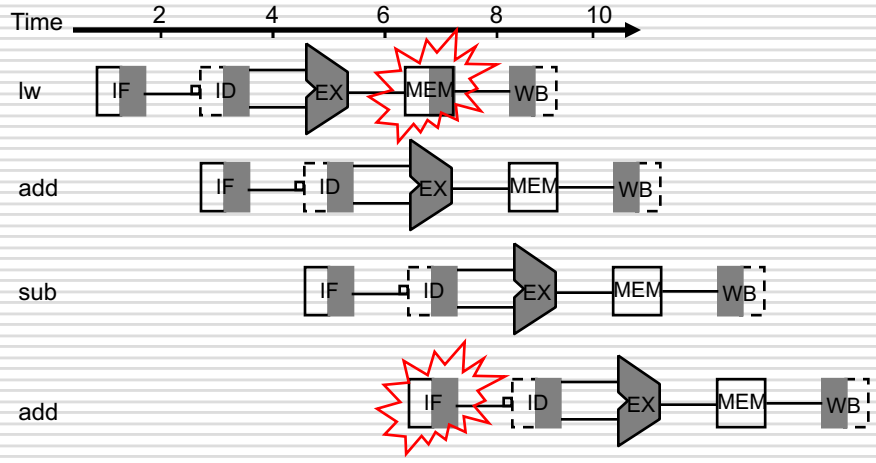
Graphically Representing Pipelines



- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

(10)

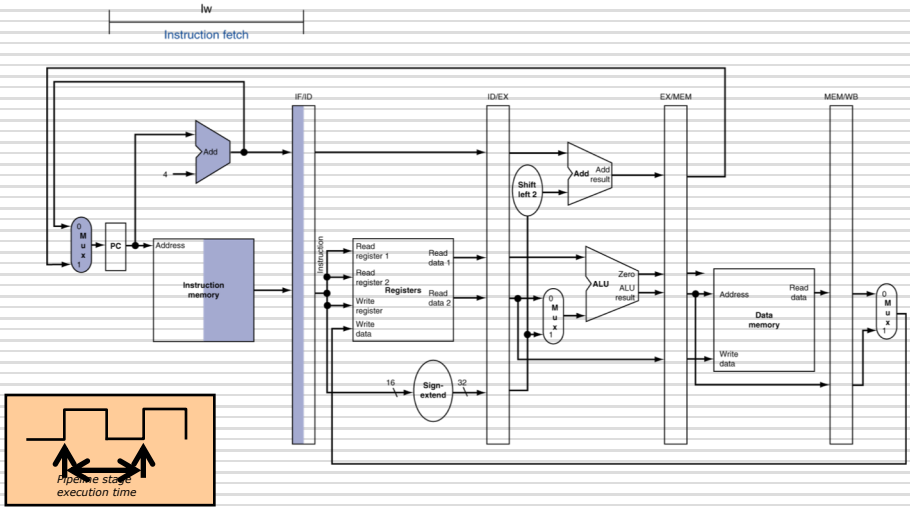
Structural Hazard



Need to separate instruction and data memory

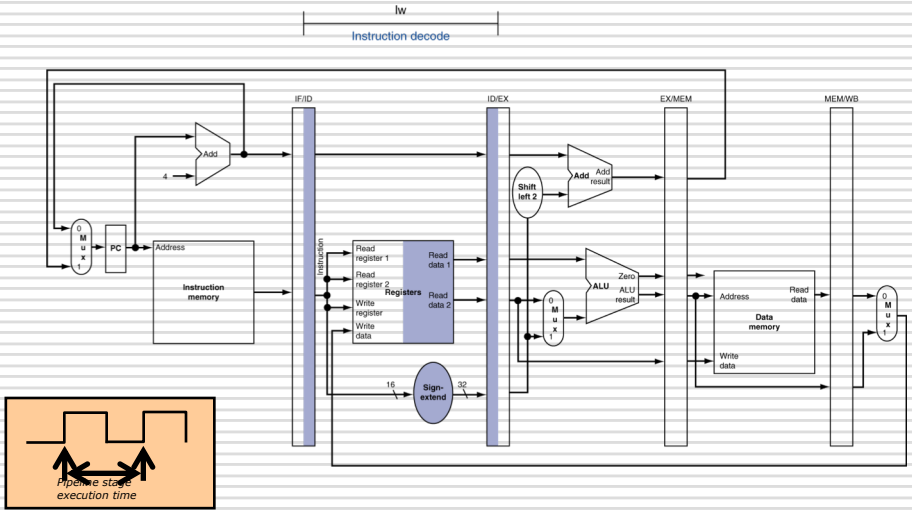
(11)

IF for Load, Store, …

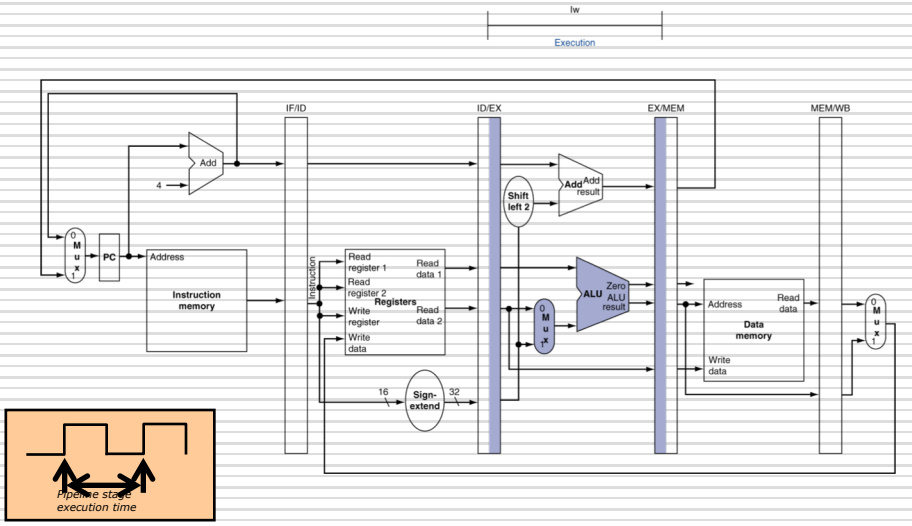


(12)

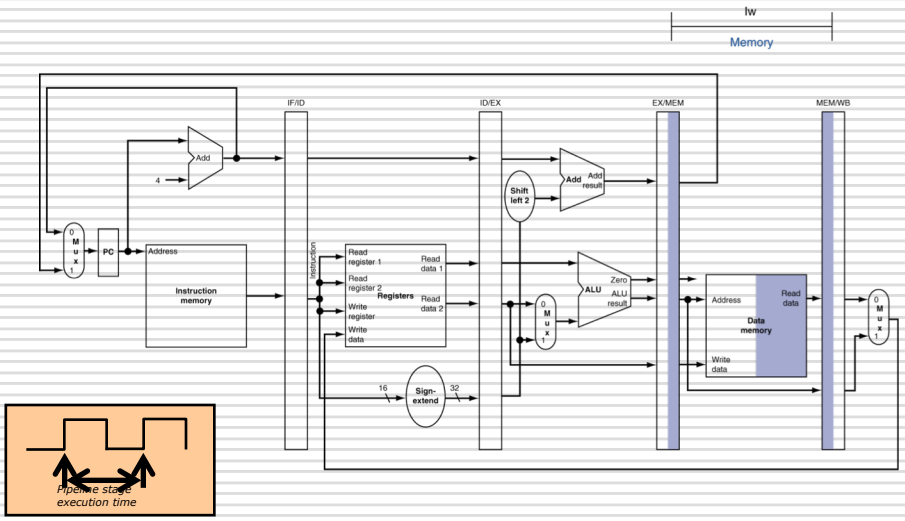
ID for Load, Store, …



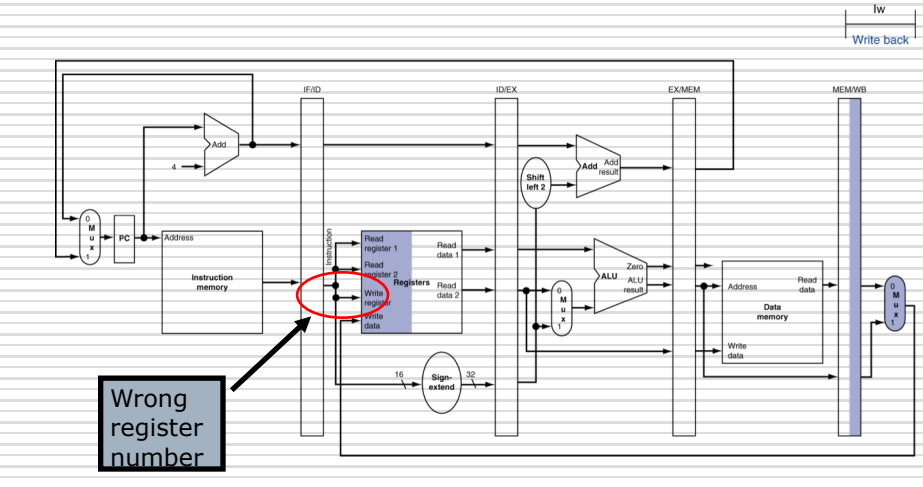
EX for Load



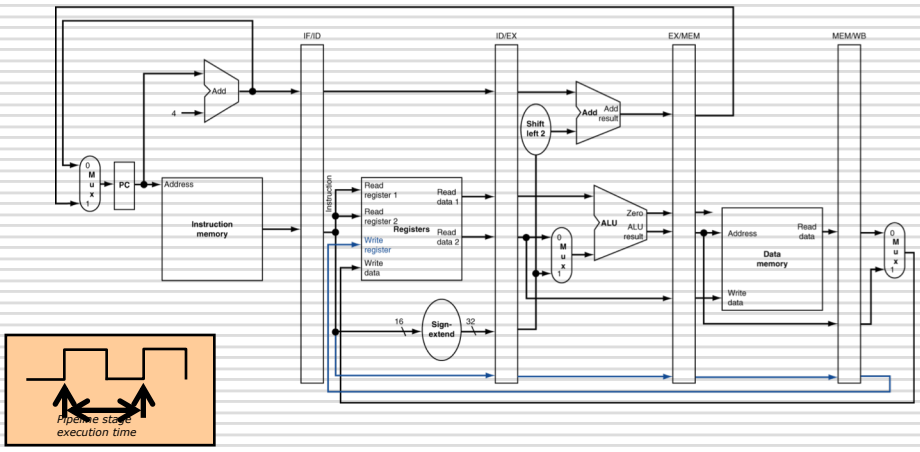
MEM for Load



WB for Load

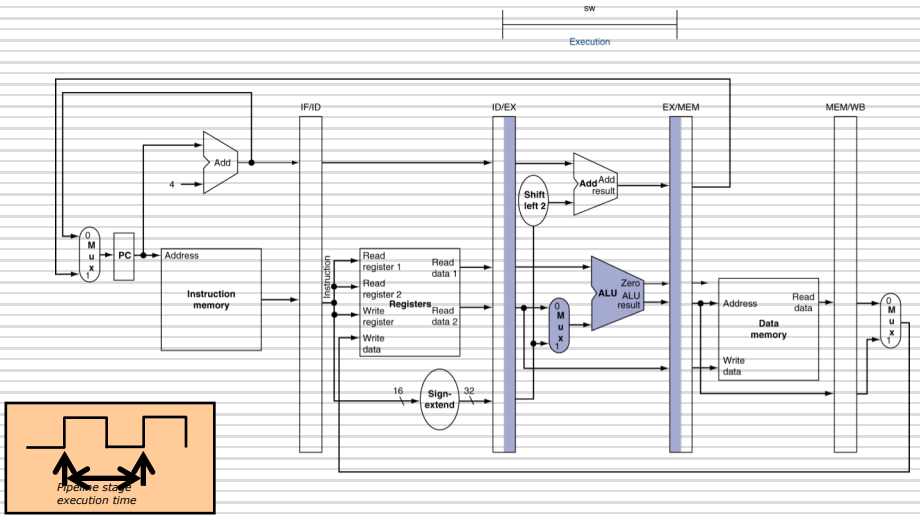


Corrected Datapath for Load



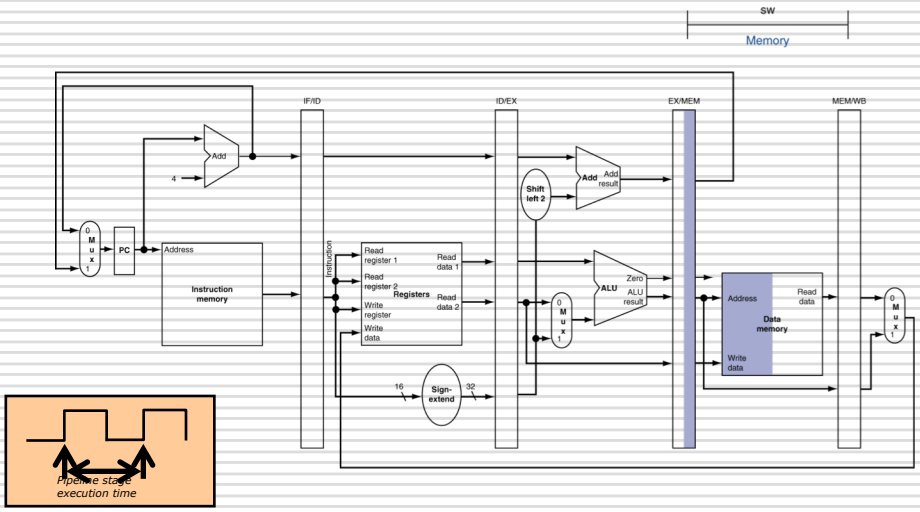
(17)

EX for Store



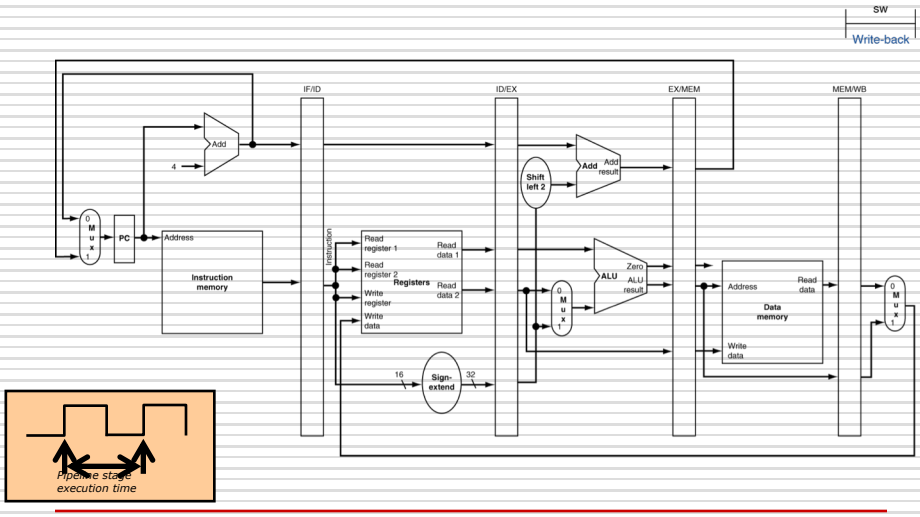
(18)

MEM for Store



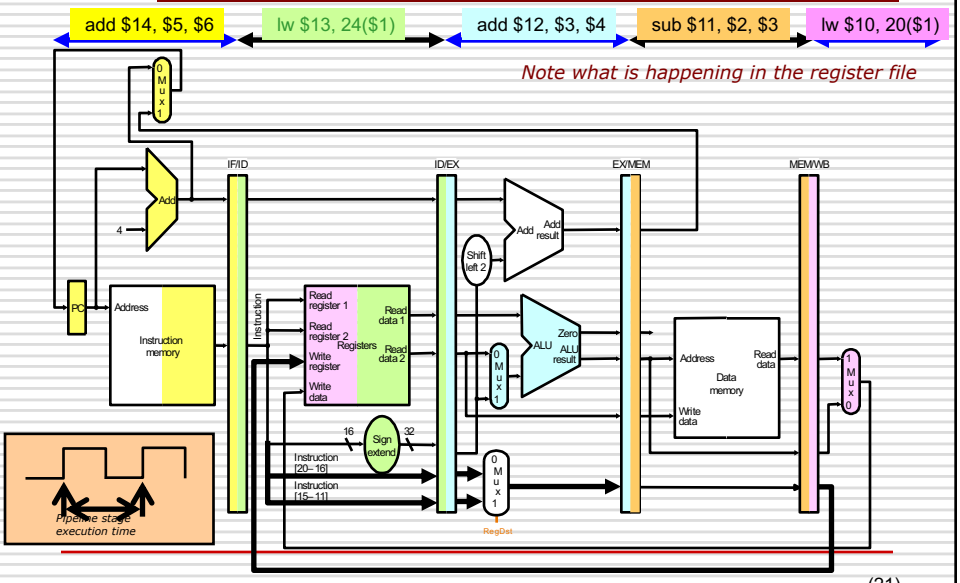
(19)

WB for Store



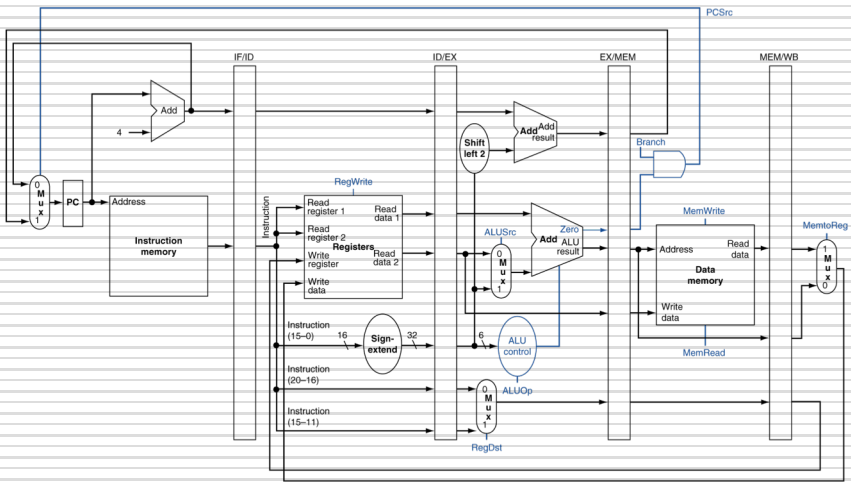
(20)

Pipelining Example



(21)

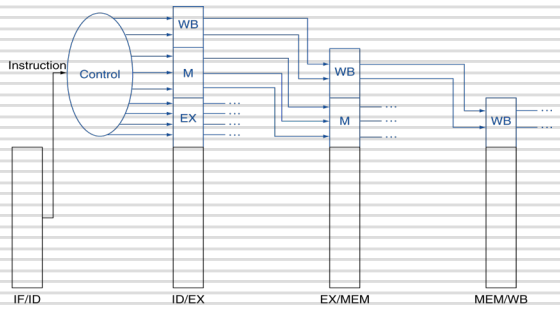
Pipelined Control (Simplified)



(22)

Pipelined Control

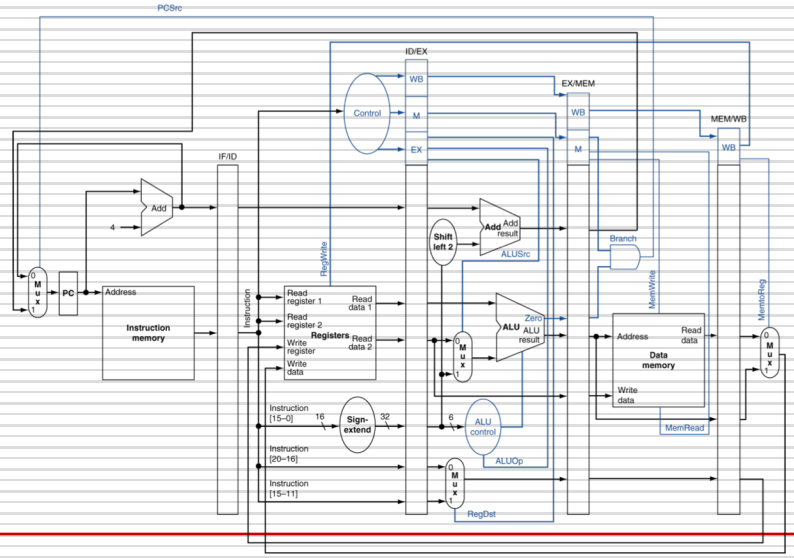
Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X



- Control signals derived from instruction
  - As in single-cycle implementation
- Pass control signals along like data

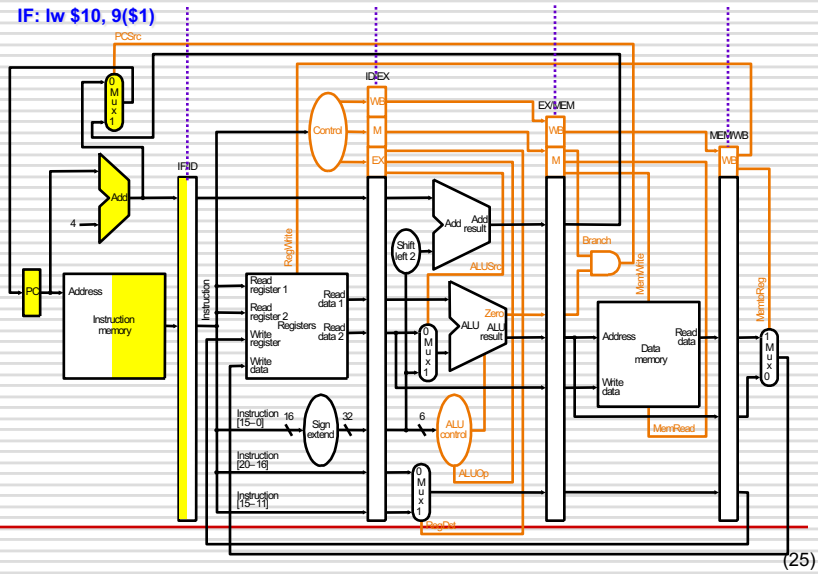
(23)

Pipelined Control



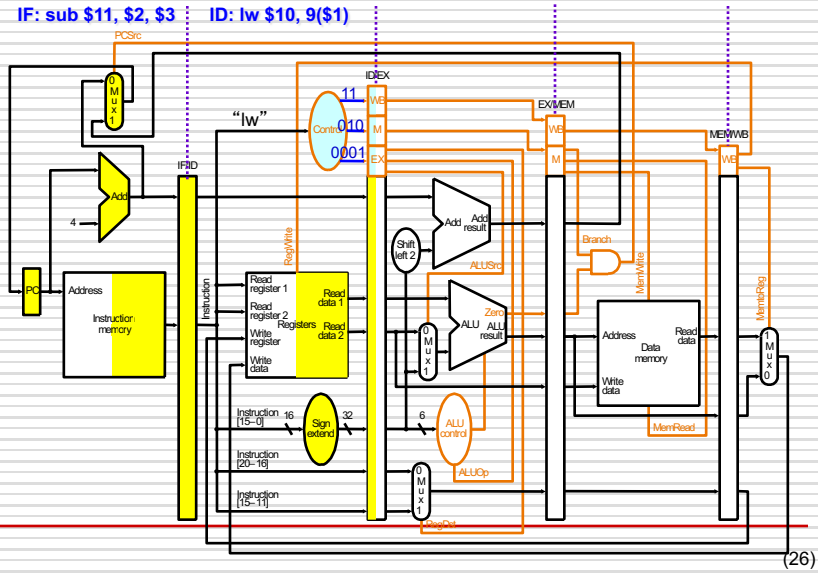
(24)

Datapath with Control



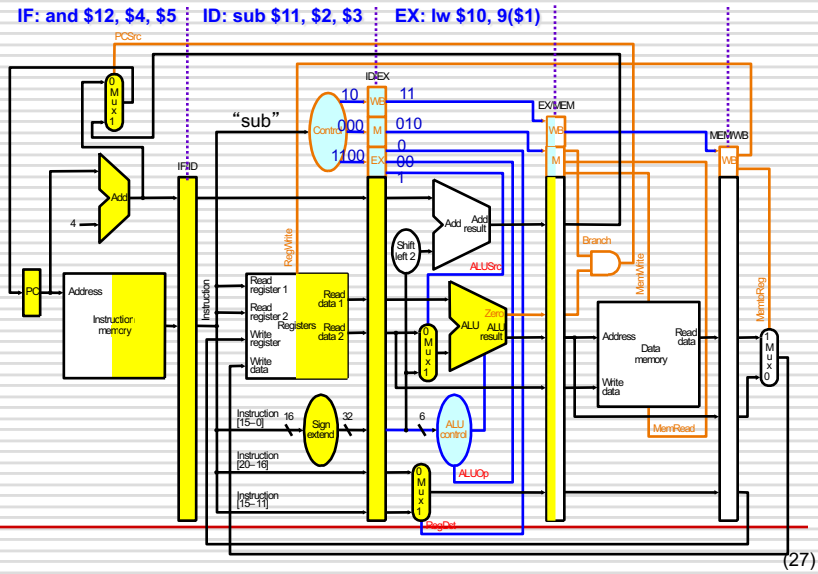
(25)

Datapath with Control



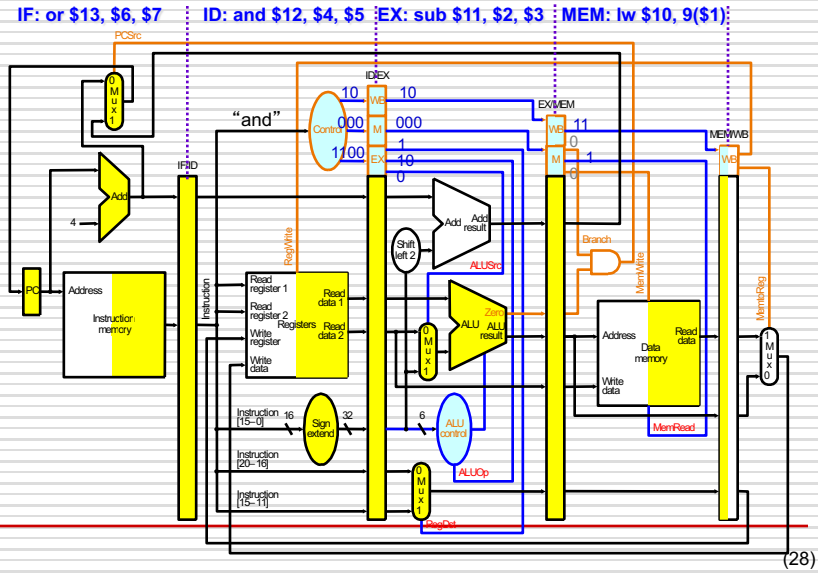
(26)

Datapath with Control



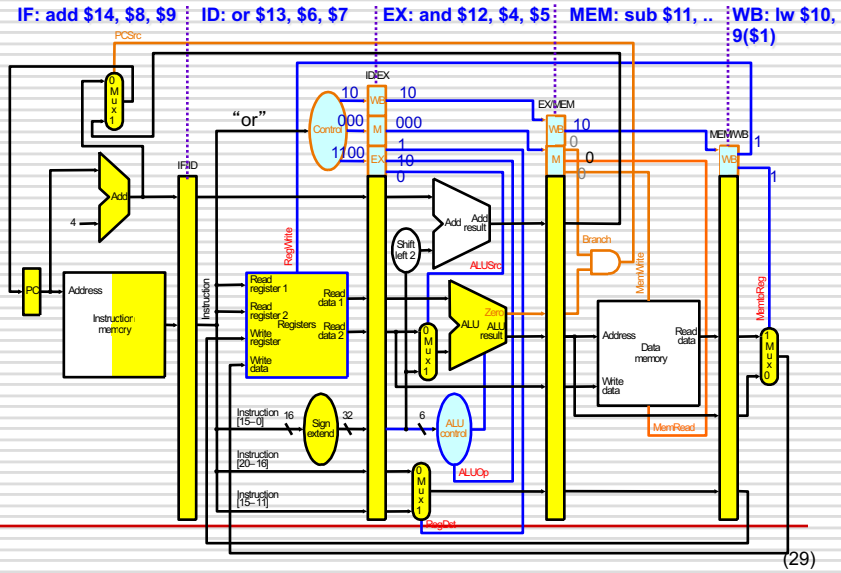
(27)

Datapath with Control

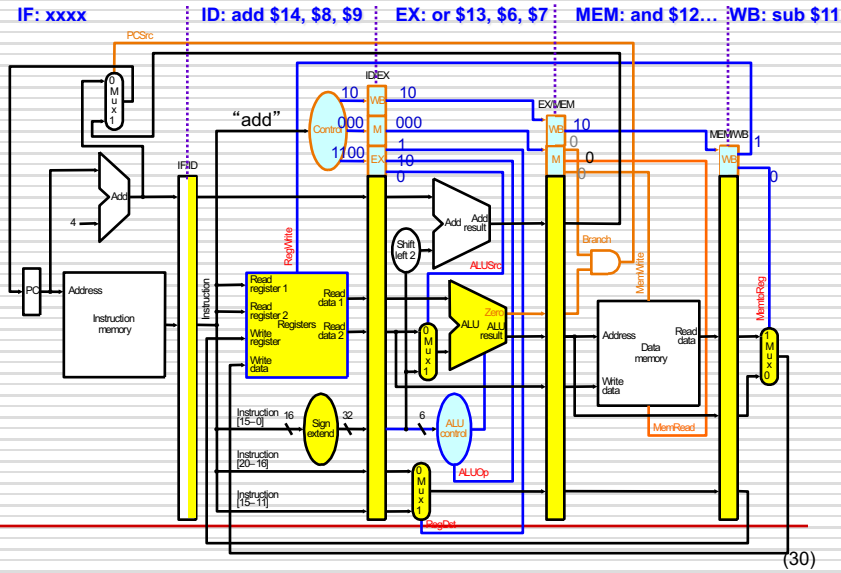


(28)

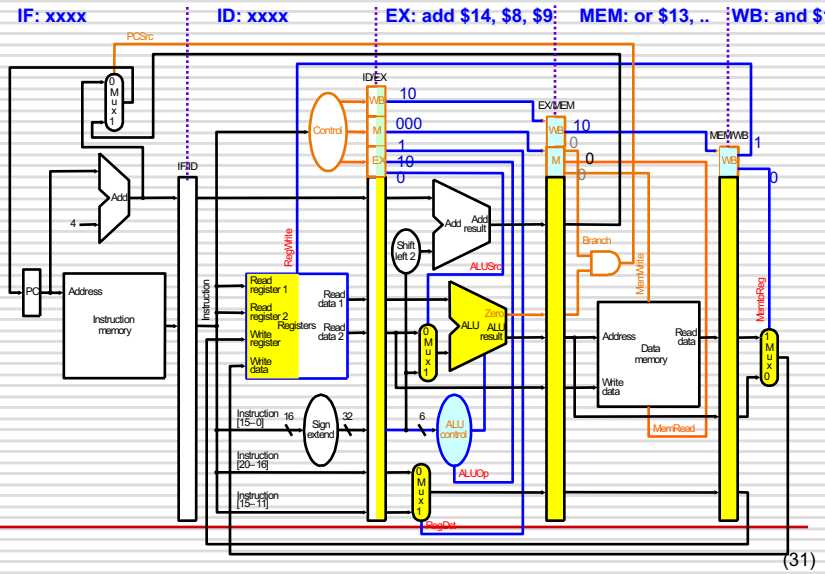
Datapath with Control



Datapath with Control

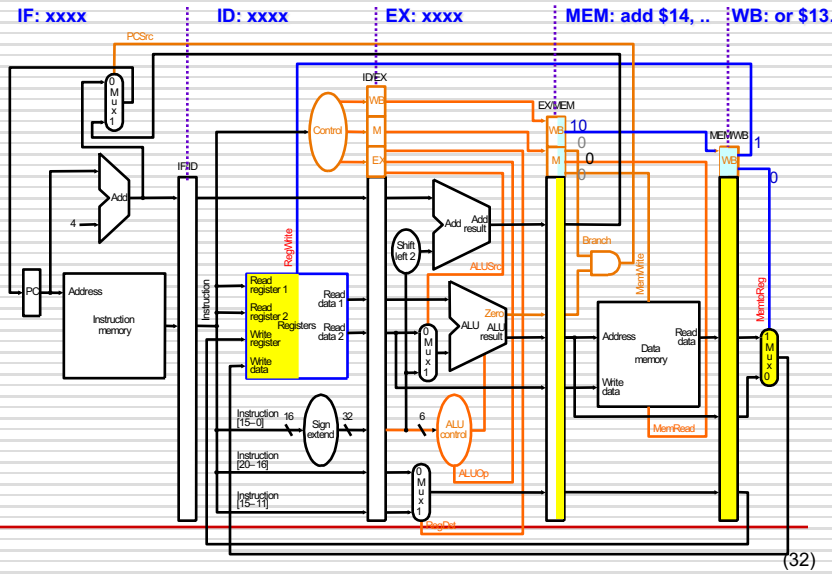


Datapath with Control



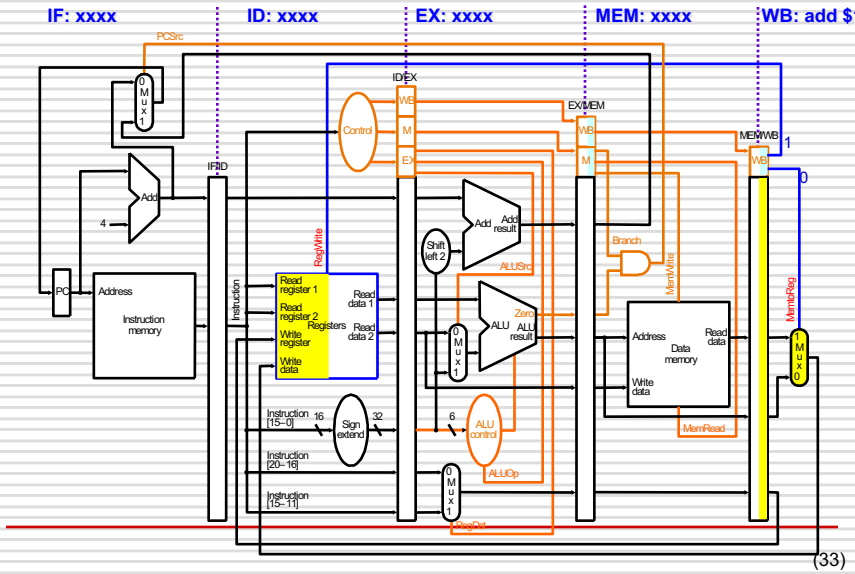
(31)

Datapath with Control



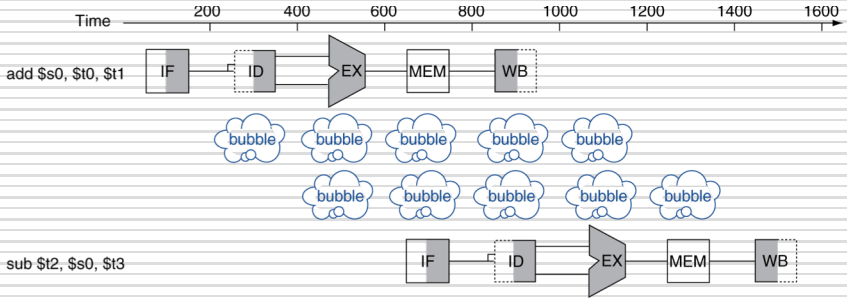
(32)

Datapath with Control



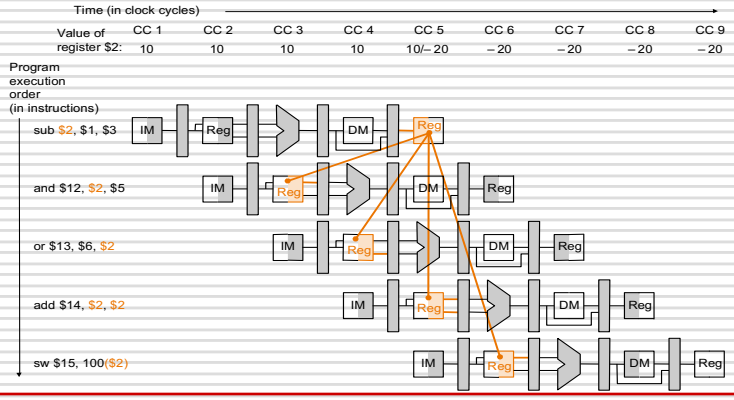
Data Hazards (4.7)

- An instruction depends on completion of data access by a previous instruction
  - `add $s0, $t0, $t1`
    `sub $t2, $s0, $t3`



Dependencies

- Problem with starting next instruction before first is finished
  - dependencies that "go backward in time" are data hazards



(35)

Software Solution

- Have compiler guarantee no hazards
- Where do we insert the "nops" ?

```
sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

- Problem: this really slows us down!

(36)

A Better Solution

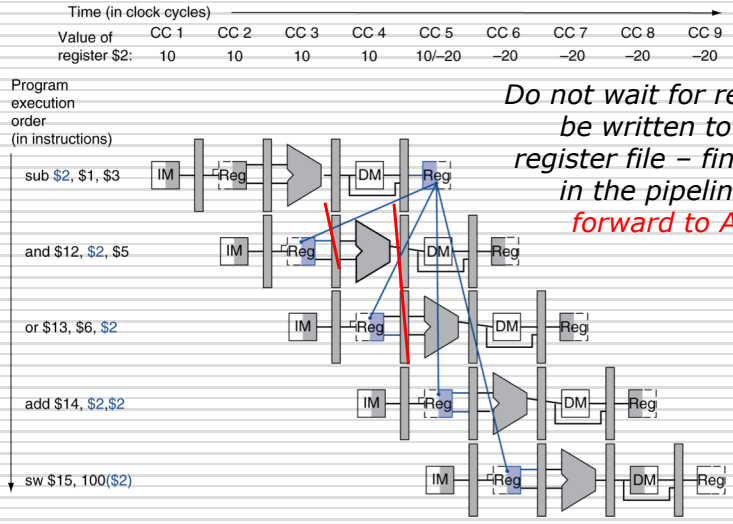
- Consider this sequence:

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

- We can resolve hazards with forwarding
  - How do we detect when to forward?

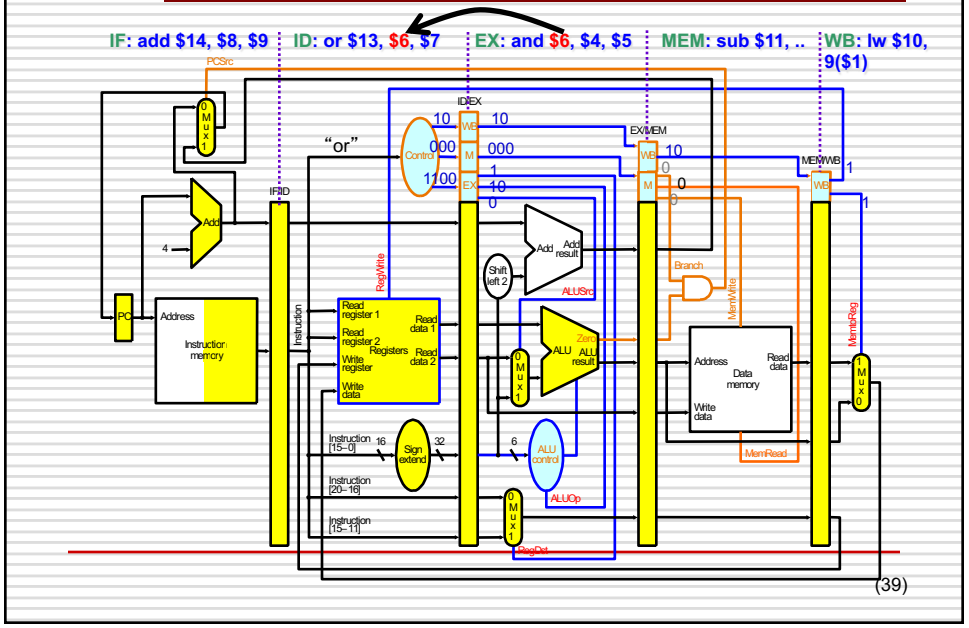
(37)

Dependencies & Forwarding

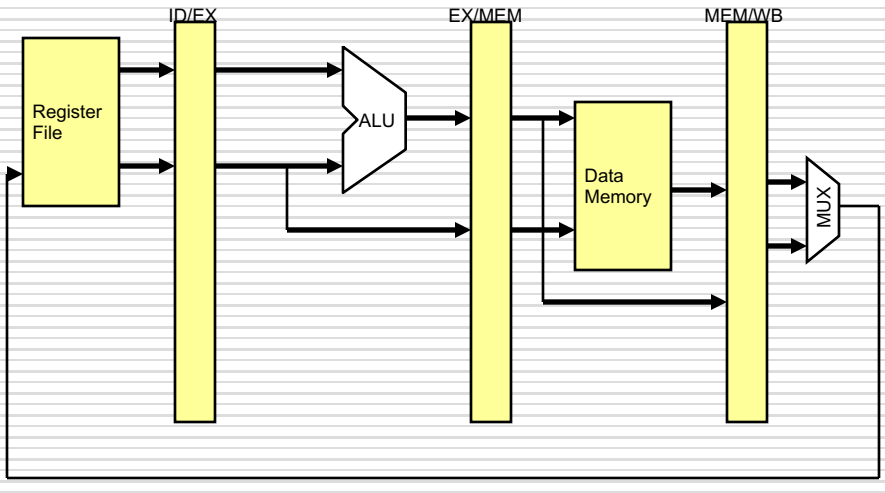


(38)

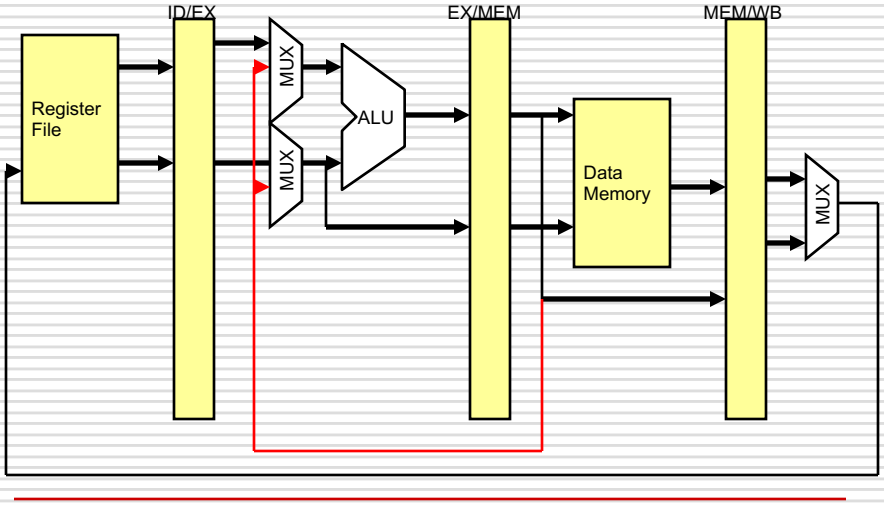
Forwarding



Forwarding (simplified)

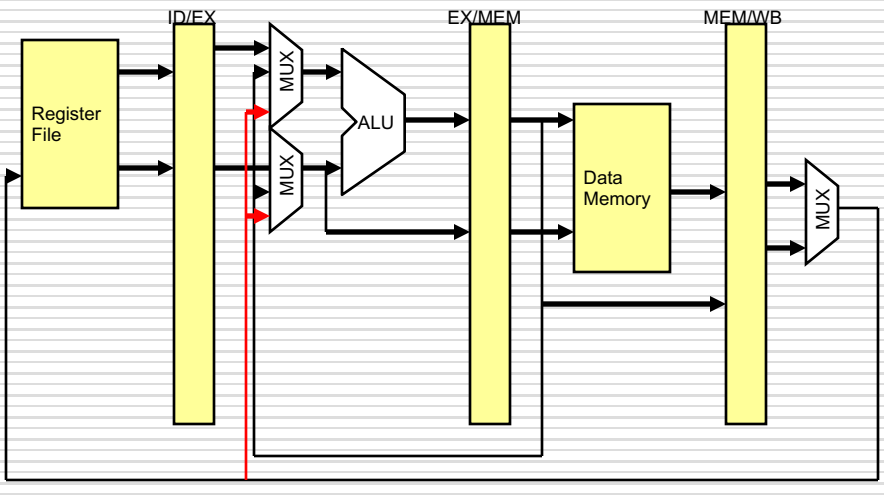


Forwarding (from EX/MEM)



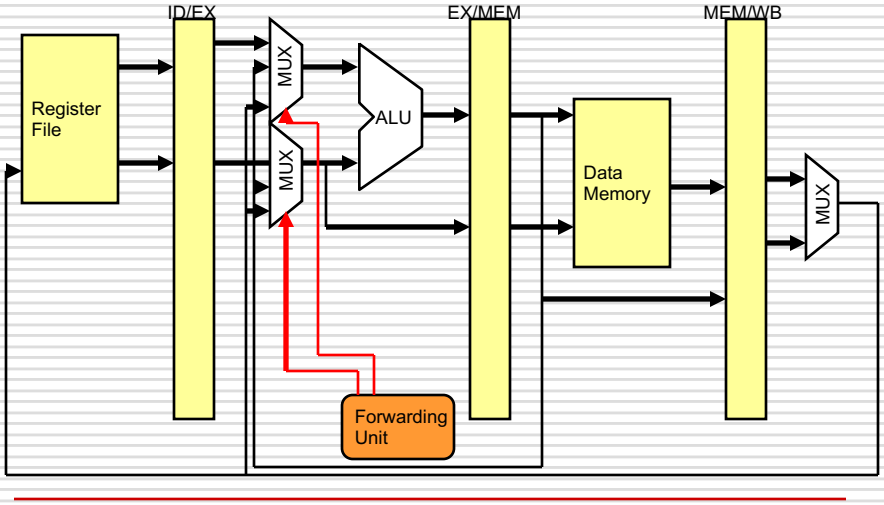
(41)

Forwarding (from MEM/WB)



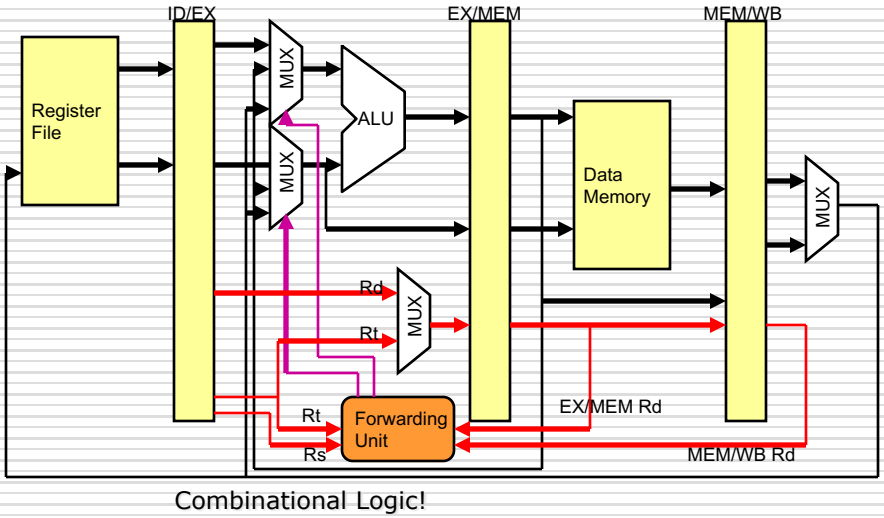
(42)

Forwarding (operand selection)



(43)

Forwarding (operand propagation)



Combinational Logic!

(44)

Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when

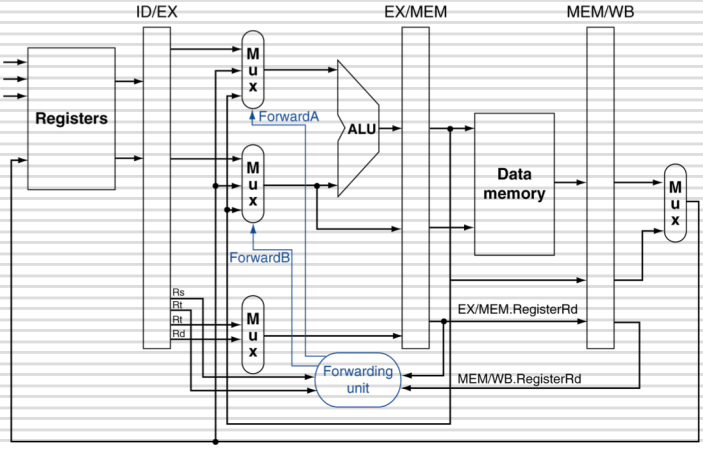
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
} Fwd from EX/MEM pipeline reg

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
} Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

Forwarding Paths



b. With forwarding

(47)

Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

(48)

Double Data Hazard

- Consider the sequence:

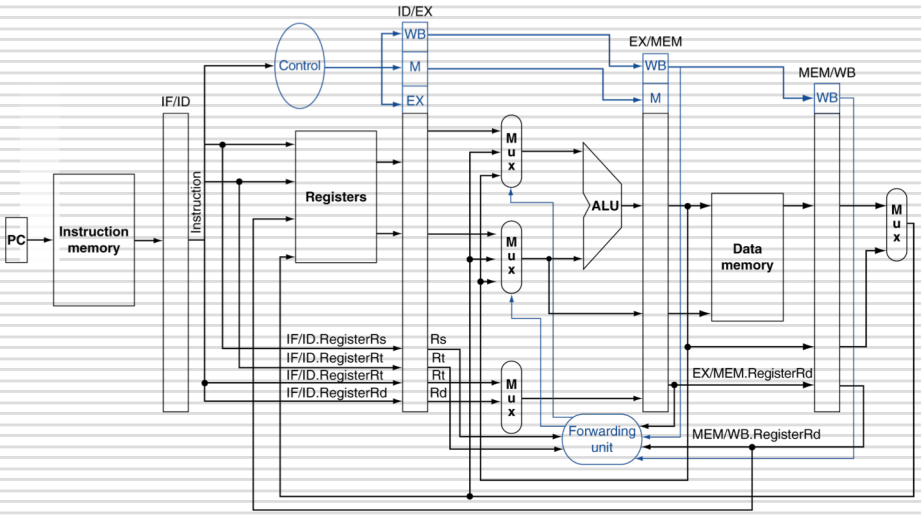
```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only forward if EX hazard condition isn't true

Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
  - Checking precedence of EX hazard

Datapath with Forwarding



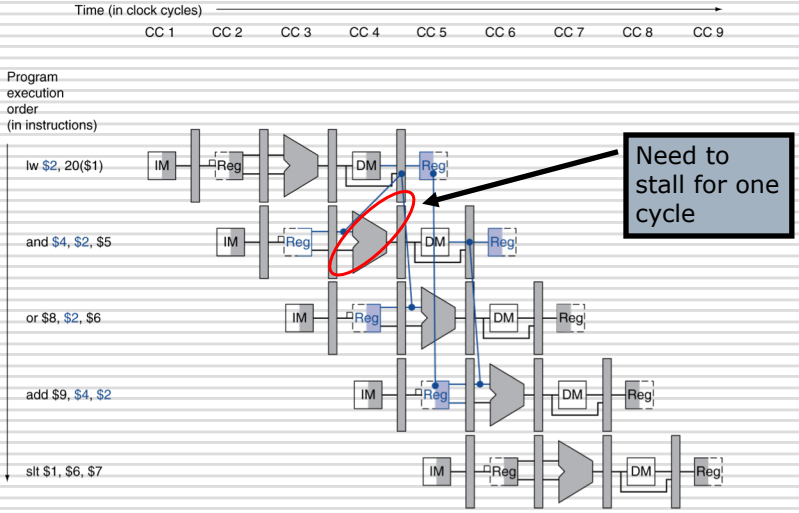
(51)

Concurrent Execution

- Correct execution is about managing dependencies
  - Producer-consumer
  - Structural (using the same hardware component)
- We will come across other types of dependencies later!

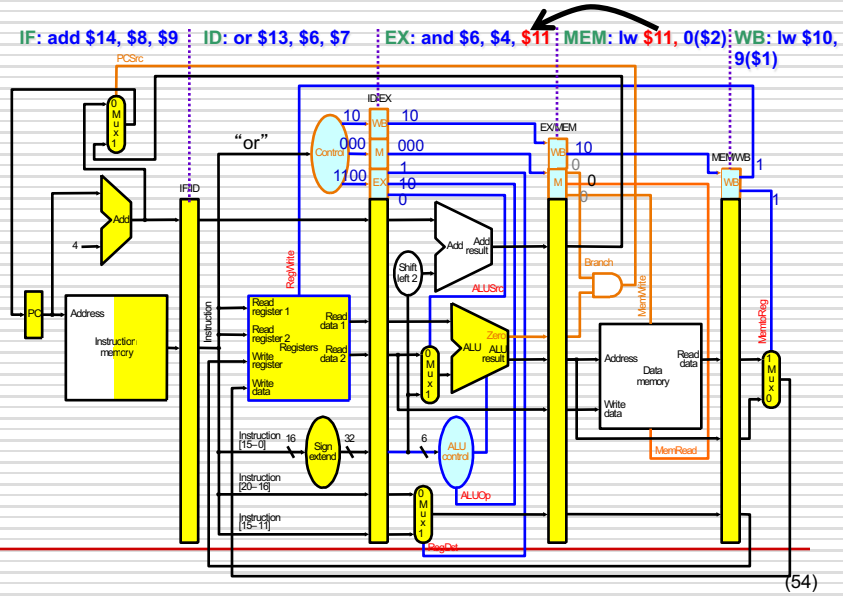
(52)

Load-Use Data Hazard



(53)

Forwarding



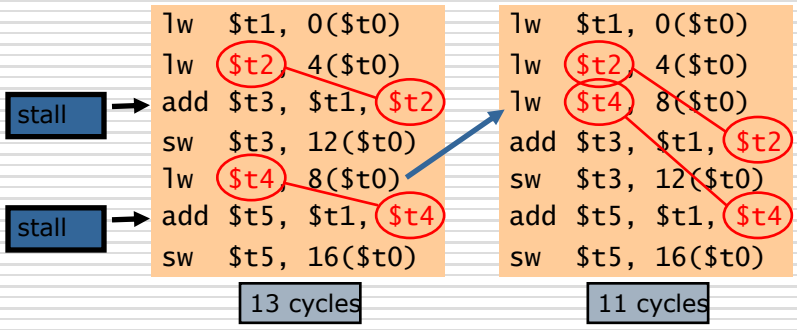
(54)

Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`

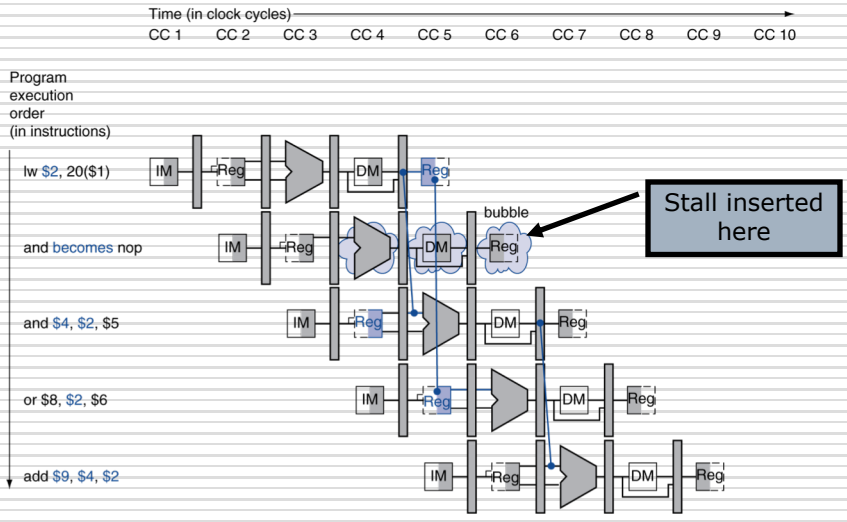


How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB perform a nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

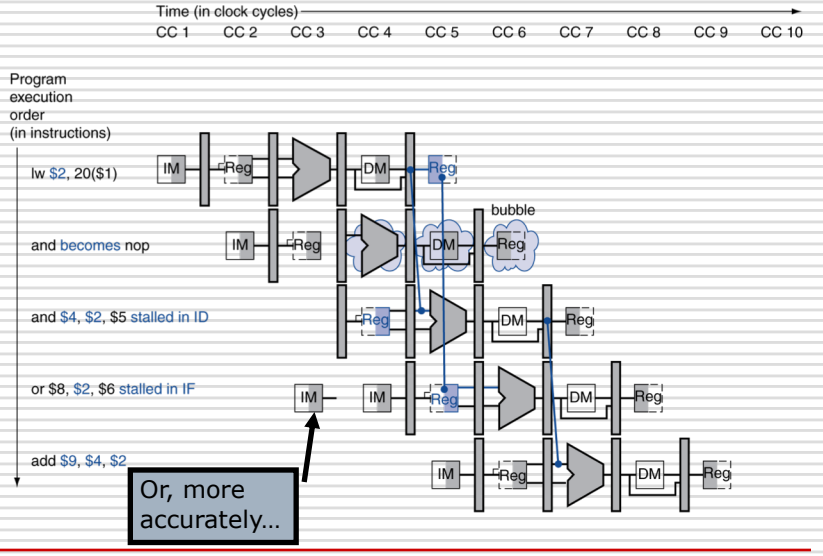
(57)

Stall/Bubble in the Pipeline



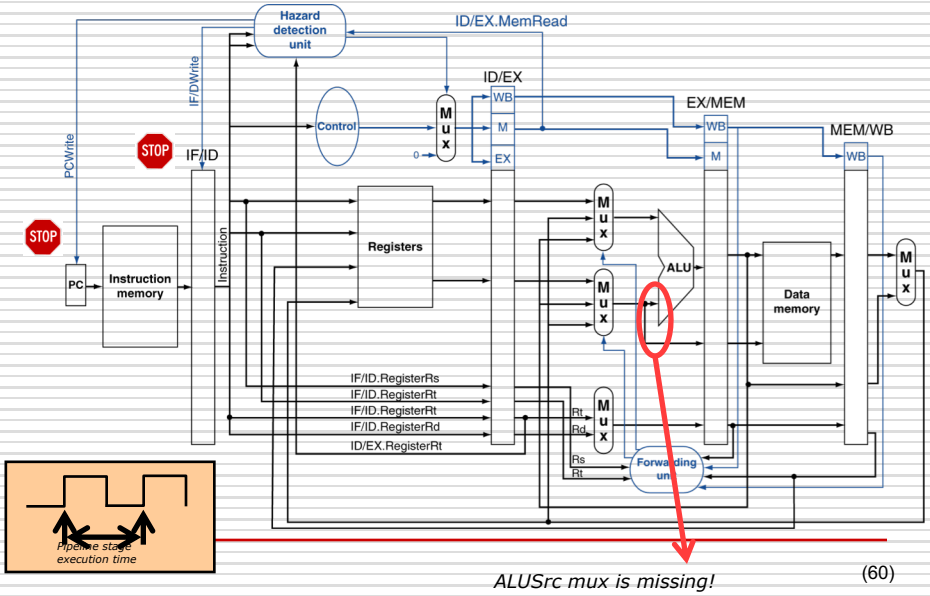
(58)

Stall/Bubble in the Pipeline



(59)

Datapath with Hazard Detection



ALUSrc mux is missing!

(60)

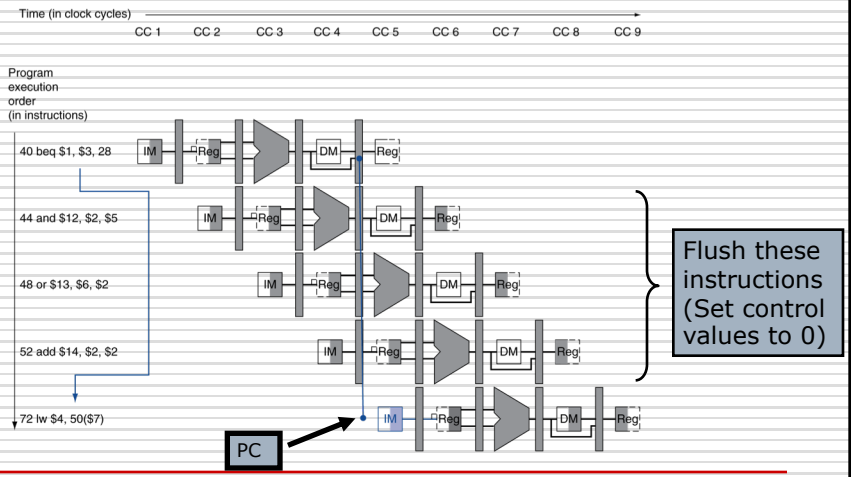
Control Hazards (4.8)

- Branch instruction determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline cannot always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and determine the branch condition

(61)

Branch Hazards

- If branch outcome determined in MEM



(62)

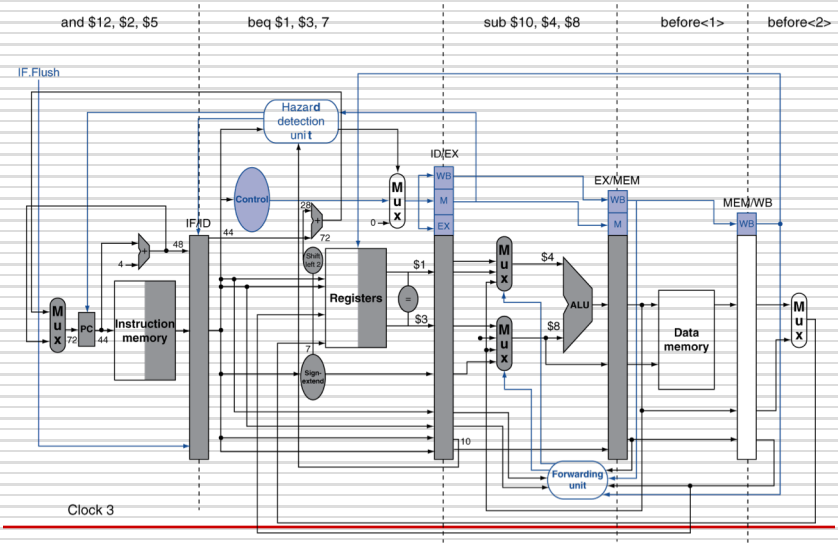
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
  - Add IF.Flush signal to squash IF/ID register
- Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 72
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
     ...
72:  lw   $4, 50($7)
```

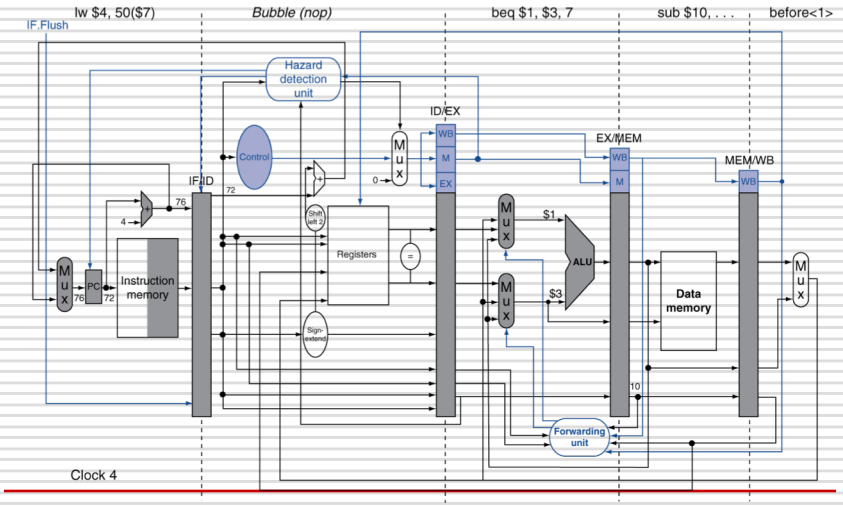
(63)

Example: Branch Taken



(64)

Example: Branch Taken

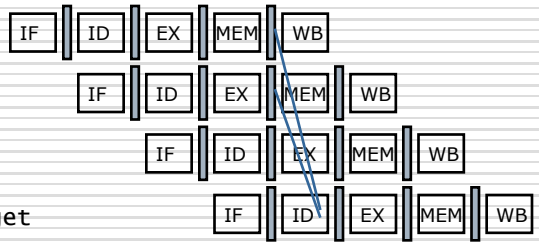


(65)

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

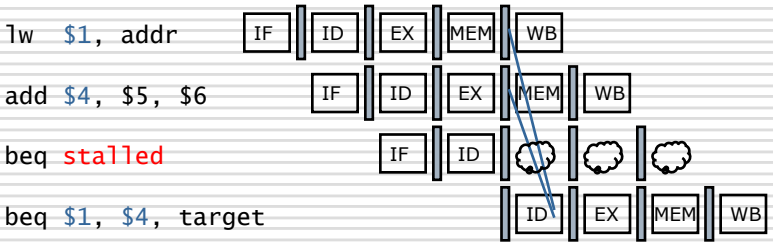


- Can resolve using forwarding to ID
  - Need to add forwarding logic!

(66)

Data Hazards for Branches

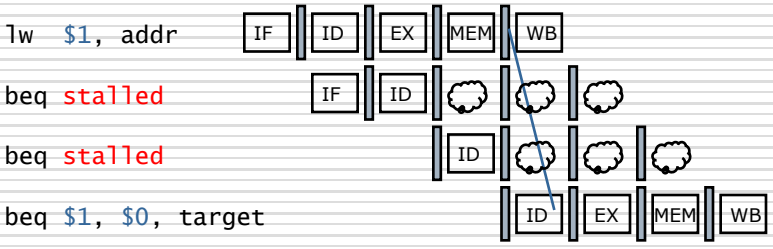
- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle



(67)

Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles



(68)

Delay Slot (MIPS)

- Expose pipeline
- Load and jump/branch entail a "delay slot"
- The instruction right after the jump or branch is executed before the jump/branch

```
jal   function_A
add   $4, $5, $6   ; executed before jmp
lw    $12, 8($4)   ; executed after return
```

- Jump/branch and the delay slot instruction are considered "indivisible"
- In the delay slot, the compiler needs to schedule
  - A useful instruction (either before the jmp, or after the jmp w/o side effect)
  - otherwise a NOP

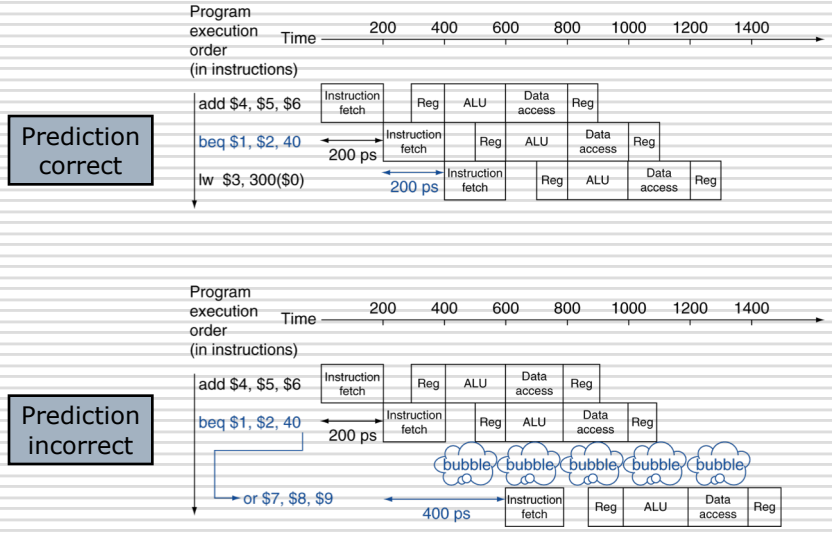
(69)

Branch Prediction

- Longer pipelines cannot readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

(70)

MIPS with Predict Not Taken



1-Bit Predictor: Shortcoming

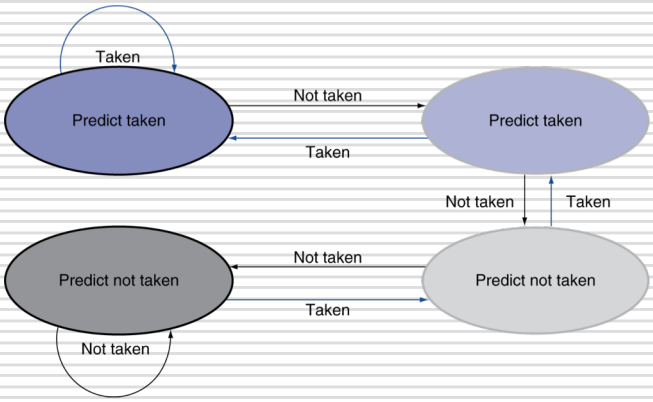
- Inner loop branches mispredicted twice!

```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor: State Machine

- Only change prediction on two successive mispredictions



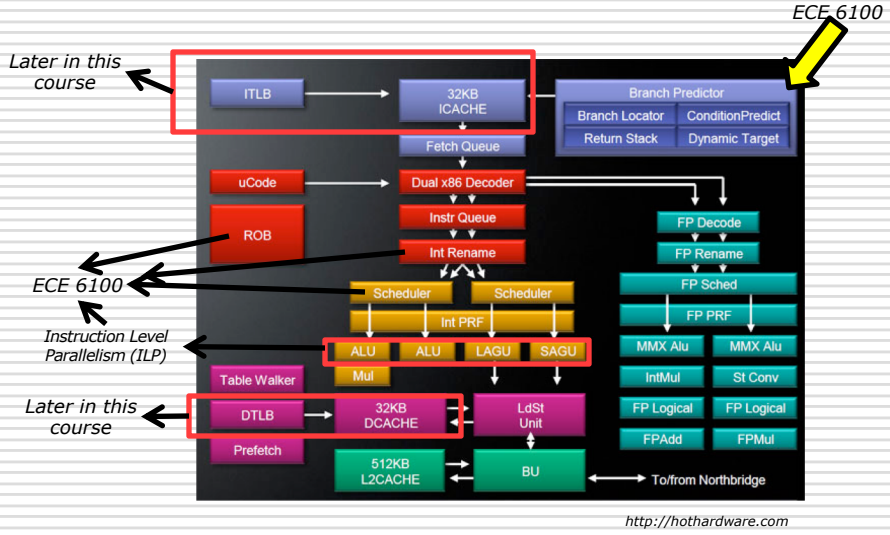
(73)

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

(74)

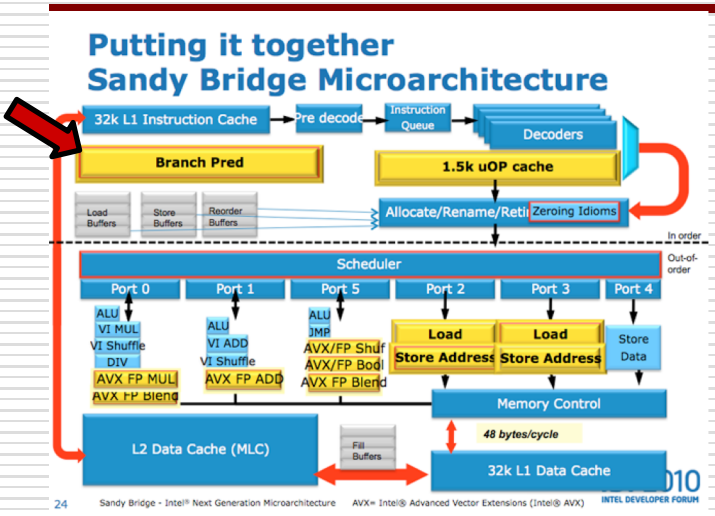
AMD Bobcat



http://hothardware.com

(75)

Intel Sandy Bridge



bdti.com

(76)

Exceptions and Interrupts (4.9)

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Updates to the data path
  - Recording the cause of the exception and transferring control to the OS
  - Consider the impact of hardware modifications on the critical path

(77)

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 80000180

(78)

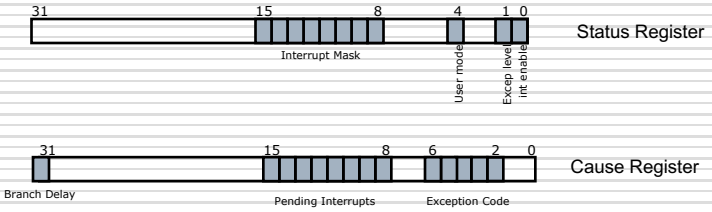
Exception Handling: Operations

- Add two registers to the datapath
  - EPC and Cause registers
- Add a state for each exception condition
  - Use the ALU to compute the EPC contents
  - Write the Cause register with exception condition
  - Update the PC with OS handler address
  - Generate control signals for each operation
- See Appendix A.7 for details of MIPS 2000/3000 implementation

(79)

The OS Interactions

- The MIPS 32 Status and Cause Registers



- Operating System handlers interrogate these registers
- Manage all state saving requirements
- Read example in A.7

(80)

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

(81)

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

(82)

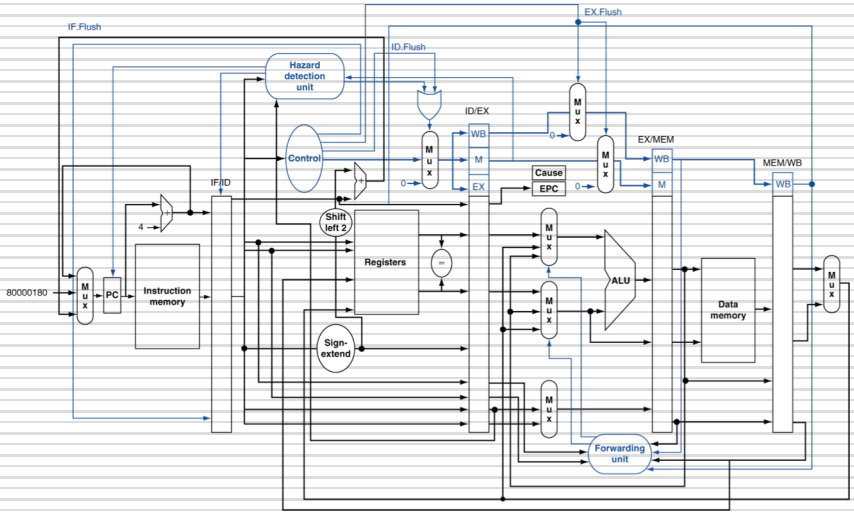
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

  `add $1, $2, $1`
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

(83)

Pipeline with Exceptions



(84)

Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Re-fetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Exception Example

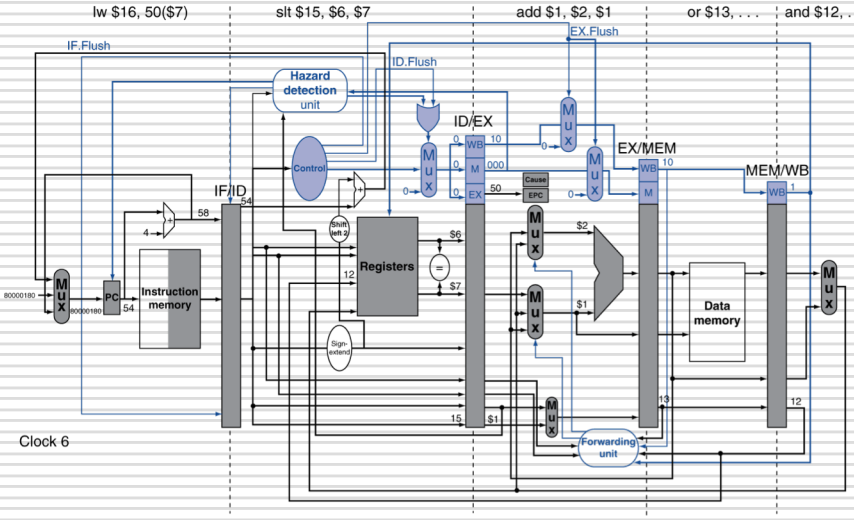
- Exception on add in

```
40    sub  $11, $2, $4
44    and  $12, $2, $5
48    or   $13, $2, $6
4C    add  $1,  $2, $1
50    slt  $15, $6, $7
54    lw   $16, 50($7)
…
```

- Handler

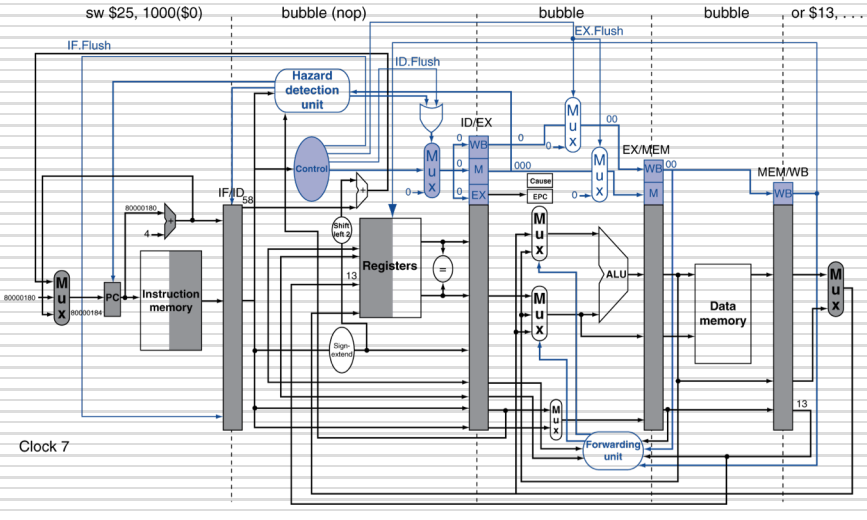
```
80000180  sw   $25, 1000($0)
80000184  sw   $26, 1004($0)
…
```

Exception Example



(87)

Exception Example



(88)

Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - "Precise" exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

(89)

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require "manual" completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

(90)

Performance

- How do we assess the impact of stall cycles?
- How close do we approach the ideal of one instruction per cycle execution time?
- Back to the CPI model!

(91)

Recall: Program Execution time

Number of instruction classes

$$ExecutionTime = \left[\sum_{i=1}^{n} C_i \times CPI_i\right] \times cycle\_time$$

~= Instruction_count * CPI_{avg} * clock_cycle_time

algorithms/compiler | architecture | technology

$$CPI_{avg} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n}\left(CPI_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}}\right)$$

Relative frequency

(92)

Study Guide

- Given a code block, and initial register values (those that are accessed) be able to determine state of all pipeline registers at some future clock cycle.
- Determine the size of each pipeline register
- Track pipeline state in the case of forwarding and branches
- Compute the number of cycles to execute a code block
- Modify the datapath to include forwarding and hazard detection for branches (this is trickier and time consuming but well worth it)

(93)

Study Guide (cont.)

- Schedule code (manually) to improve performance, for example to eliminate hazards and fill delay slots
- Modify the data path to add new instructions such as j
- Modify the data path to accommodate a two cycle data memory access, i.e., the data memory itself is a two cycle pipeline
  - Modify the forwarding and hazard control logic
- Given a code sequence, be able to compute the number of stall cycles

(94)

Study Guide (cont.)

- Track the state of the 2-bit branch predictor over a sequence of branches in a code segment, for example a for-loop
- Show the pipeline state before and after an exception has taken place.

(95)

Glossary

- Branch prediction
- Branch hazards
- Branch delay Control hazard
- Data hazard
- Delay slot
- Dynamic instruction issue
- Forwarding
- Imprecise exception
- Instruction scheduling
- Instruction level parallelism (ILP)
- Load-to-use hazard
- Pipeline bubbles
- Stall cycles
- Static instruction issue
- Structural hazard

(96)