CPU Design

ARM Assembly Language

Type	Instruction	Example	Meaning
DP	Add	`ADD r1,r2,r3`	`r1=r2+r3`
DP	Subtract	`SUB r1,r2,r3`	`r1=r2-r3`
DT	Load reg	`LDR r1,[r2,#20]`	`r1=M[r2+20]`
DT	Store reg	`STR r1,[r2,#20]`	`M[r2+20]=r1`
DT	Move reg	`MOV r1,r2`	`r1=r2`
DP	AND	`AND r1,r2,r3`	`r1=r2&r3`
DP	OR	`ORR r1,r2,r3`	`r1=r2|r3`
B	Compare	`CMP r1,r2`	`cond flag=r1-r2`
B	EQ/LT/GT/NE/...	`BEQ L1`	`goto L1+PC+8 if equal`
B	Branch	`B L1`	`goto L1`
B	Branch & link	`BL L1`	`goto L1`

MIPS Assembly Language

Type	Instruction	Example	Meaning
DP	Add	`add r1,r2,r3`	`r1=r2+r3`
DP	Subtract	`sub r1,r2,r3`	`r1=r2-r3`
DT	Load reg	`lw r1,[r2,#20]`	`r1=M[r2+20]`
DT	Store reg	`sw r1,[r2,#20]`	`M[r2+20]=r1`
DP	AND	`and r1,r2,r3`	`r1=r2&r3`
DP	OR	`or r1,r2,r3`	`r1=r2|r3`
B	BEQ	`beq r1,r2,L1`	`goto L1+PC+4 if equal`
B	BNE	`bne r1,r2,L1`	`goto L1+PC+4 if not equal`
DP	Set on LT	`slt r1,r2,r3`	`r2<r3? r1=1 : r1=0`
B	Jump	`j L1`	`goto L1`
B	Jump & link	`jal L1`	`goto L1`

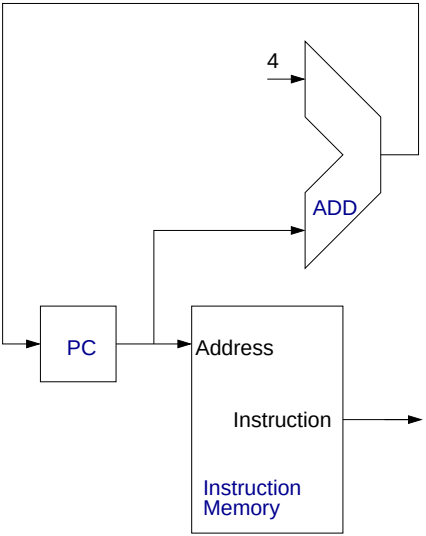
Introduction

- CPU performance factor
  - Instruction count - Determined by ISA and compiler
  - CPI and cycle time
- Will examine basic implementation
  - Memory reference instruction - `lw, sw`
  - Arithmetic-logical instruction - `add, sub, and, or, slt`
  - Branch instructions - `beq, j`
    - `beq R0, R1, offset`
    - `j offset`

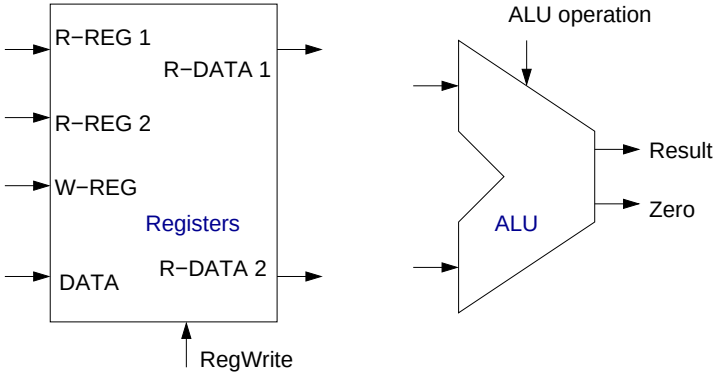
Instruction Execution

- PC – Instruction memory, fetch instruction
- Registers file
- Depending on instruction class
  - Use ALU to calculate
    - ⋆ Arithmetic results
    - ⋆ Branch target address
    - ⋆ Memory address for load/store
  - Load/store data from memory
  - PC ← target address or PC=PC+4

Instruction Fetch

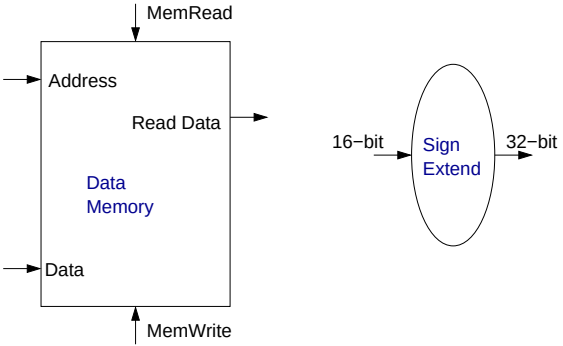


Data processing instruction



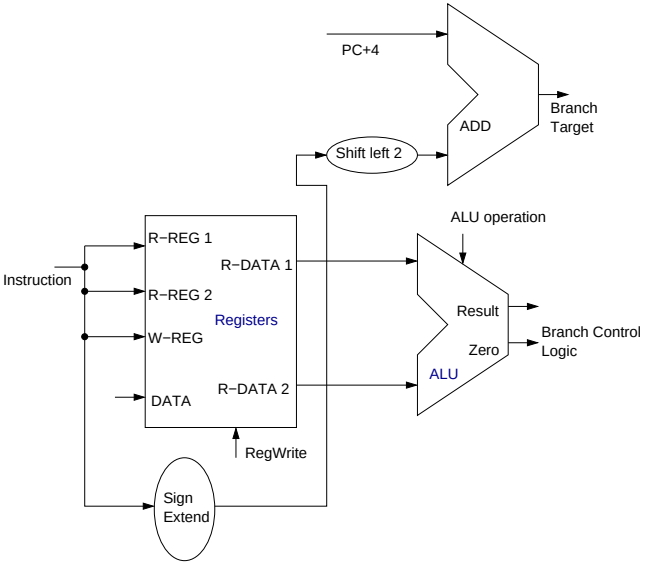
- Read two register operands
- Perform arithmetic operation
- Write results

Load/Store Instruction



- Read register operand
- Calculate address using offset
- Load: Read memory and update register
- Store: Read register and update memory

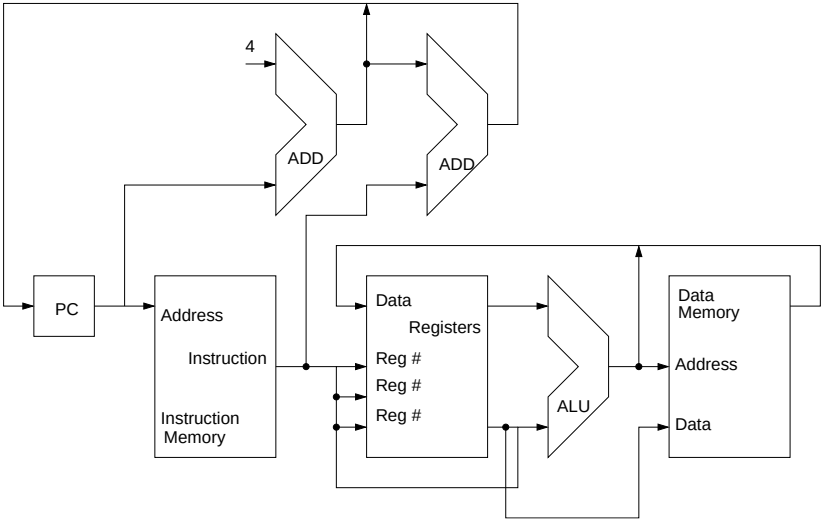
Branch Instruction



Branch Instructions

- Read register operands
- Compare operands – Use ALU, subtract and check Zero output
- Calculate target address
  - Sign extend displacement
  - Shift left by 2 for word alignment
  - Add to PC+4

CPU structure



ARM vs MIPS : Instruction set

	ARM	MIPS
Instruction size	32	32
Address space	32 bits	32 bits
Data alignment	Aligned	Aligned
Data addressing mode	9	3
Integer registers	15 GPR $\times$ 32 bits	31 GPR $\times$ 32 bits

ARM vs MIPS : Addressing mode

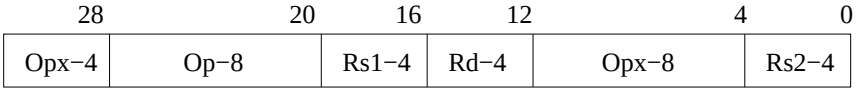
	ARM	MIPS
Register operand	×	×
Immediate operand	×	×
Register + offset	×	×
Register + register	×	–
Register + scaled register	×	–
Register + offset and update register	×	–
Register + register and update register	×	–
Auto-increment, auto-decrement	×	–
PC-relative data	×	–

ARM vs MIPS : Register files

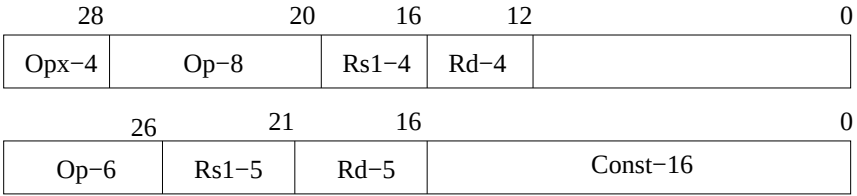
MIPS	ARM
• $0 (zero) – Wired to zero value • $1 ($at) – Reserved for assembler • $2,$3 ($v0,$v1) – Returned values • $4-$7 ($a0-$a3) – Arguments • $8-$15 ($t0-$t7) – Temporary • $16-$23 ($s0-$s7) – Saved • $24,$25 ($t8-$t9) – Temporary • $26,$27 ($k0-$k1) – Reserved for OS kernel • $28 ($gp) – Global pointer • $29 ($sp) – Stack pointer • $30 ($fp,$s8) – Frame pointer • $31 ($ra) – Linker register	• R0-R3 – Scratch registers • R4-R12 – General purpose • R13 – Stack pointer • R14 – Link register • R15 – Program counter

ARM vs MIPS : Encoding

- Register-Register

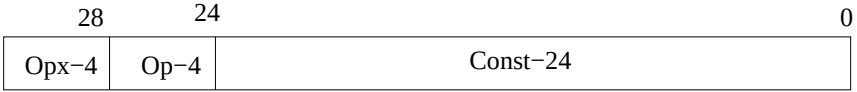


- Data transfer

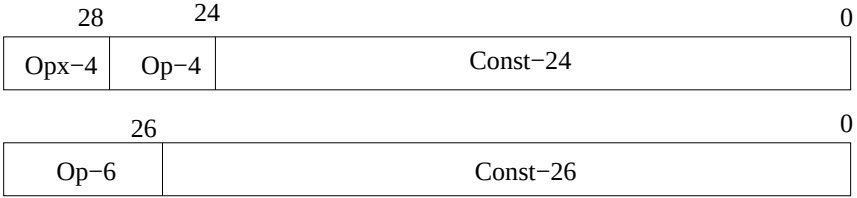


ARM vs MIPS : Encoding

- Branch



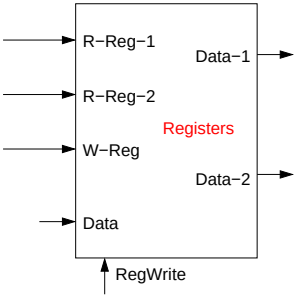
- Jump

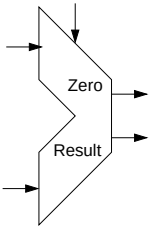


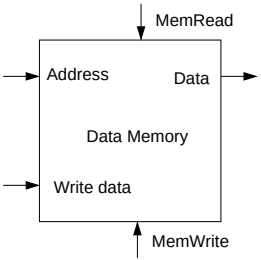
Composing the elements

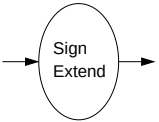
- Assuming, an instruction is executed in one cycle
- Each datapath element can only do one function at a time
- Need separate instruction and data memory
- Use multiplexers where alternate data sources are used for different instructions

Data processing and Data transfer

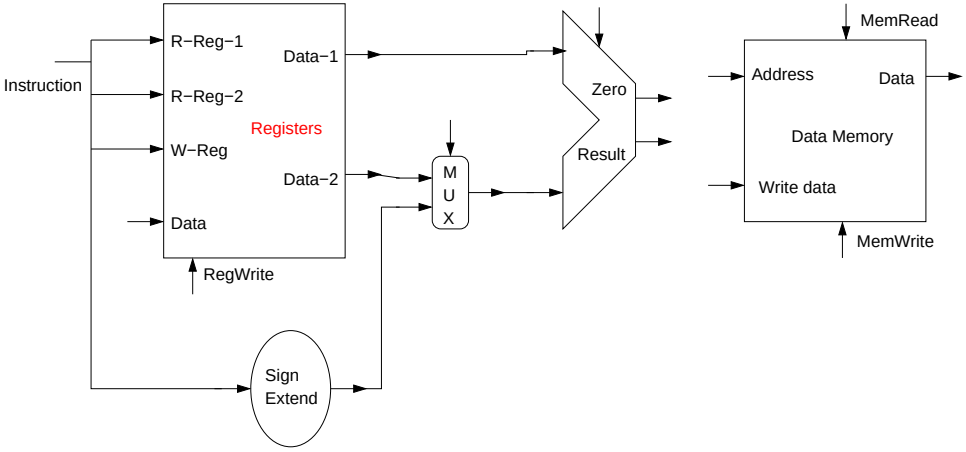




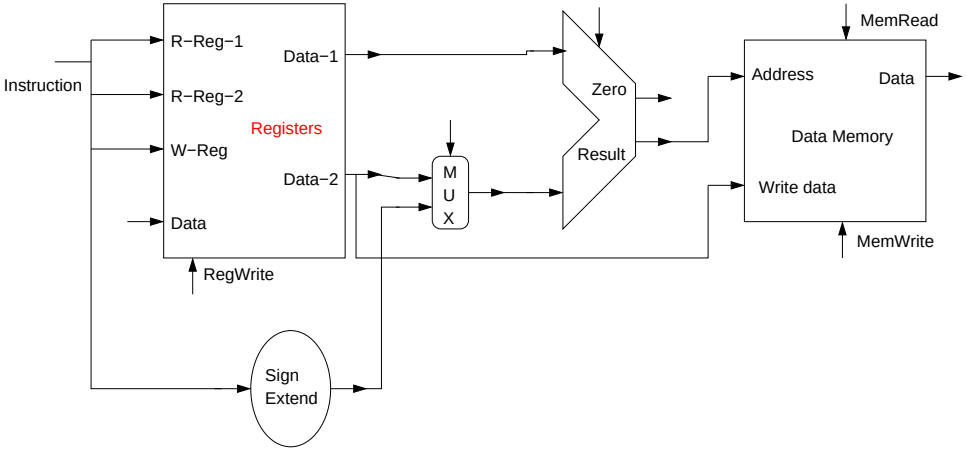




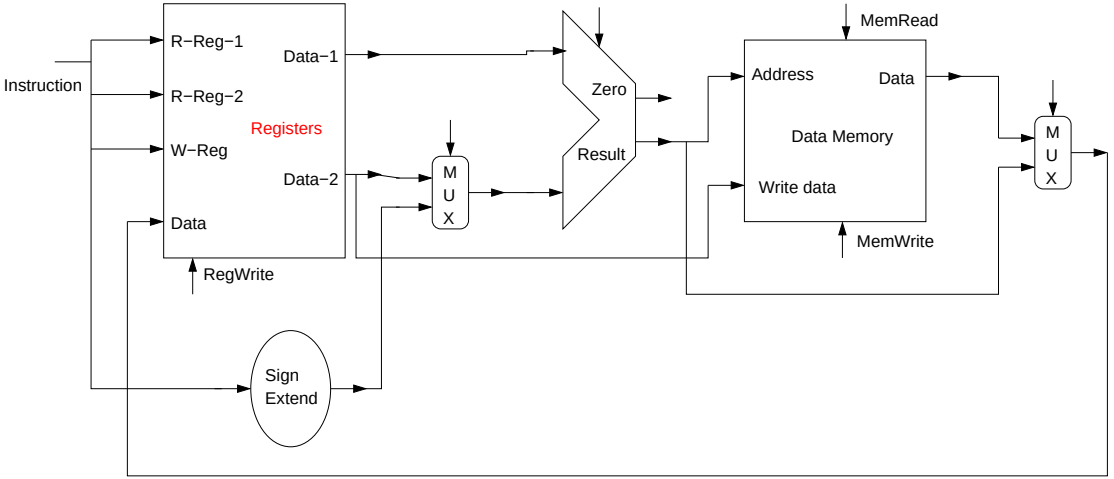
Data processing and Data transfer



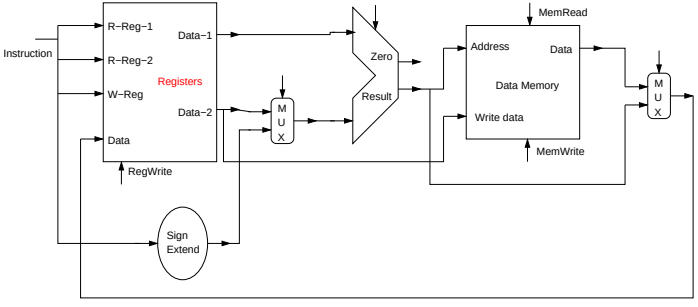
Data processing and Data transfer



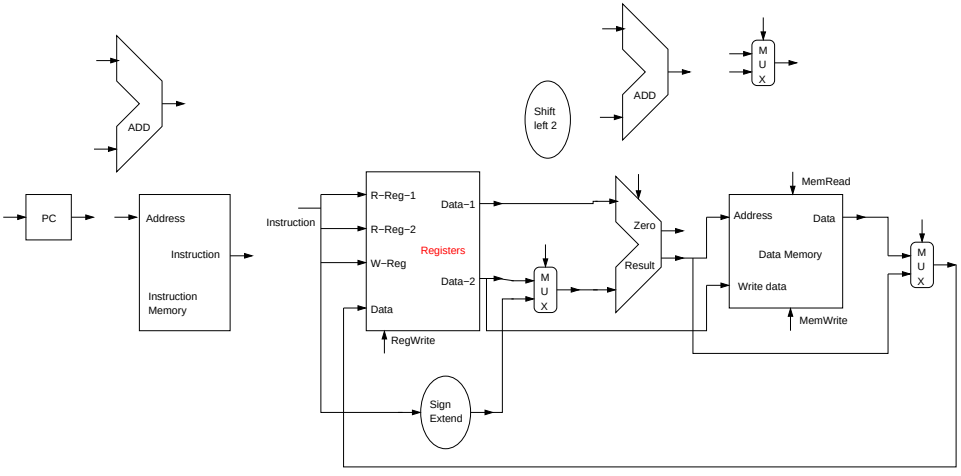
Data processing and Data transfer



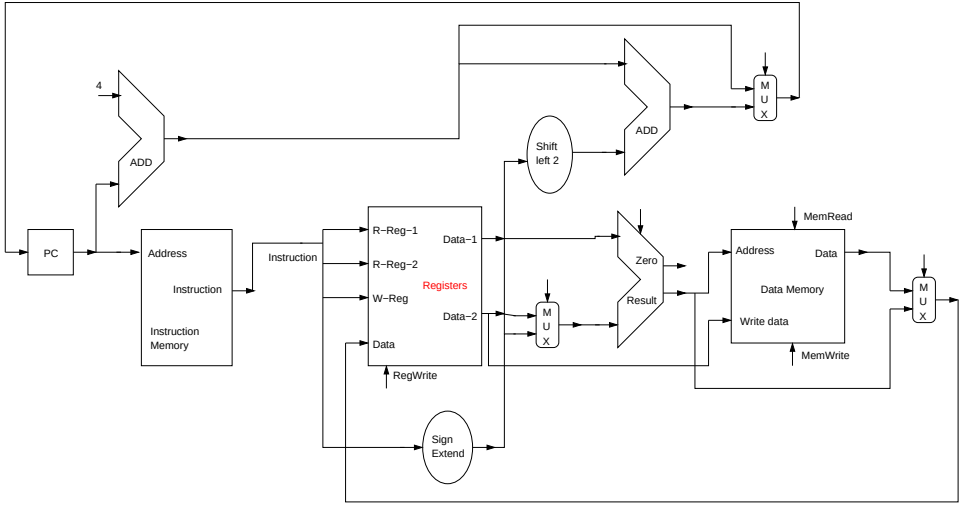
Full Datapath



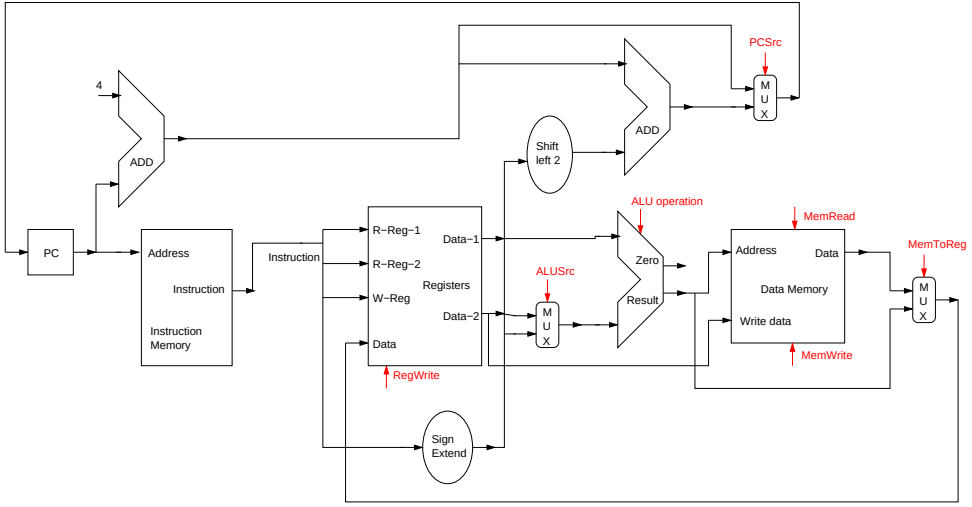
Full Datapath



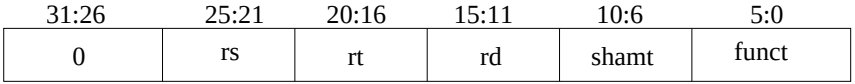
Full Datapath



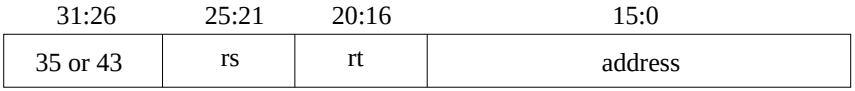
Full Datapath & Control signals



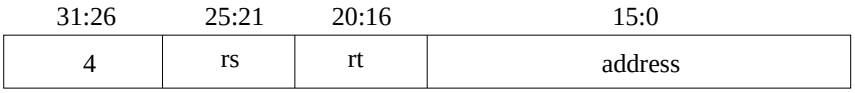
Instruction format



Data processing instructions

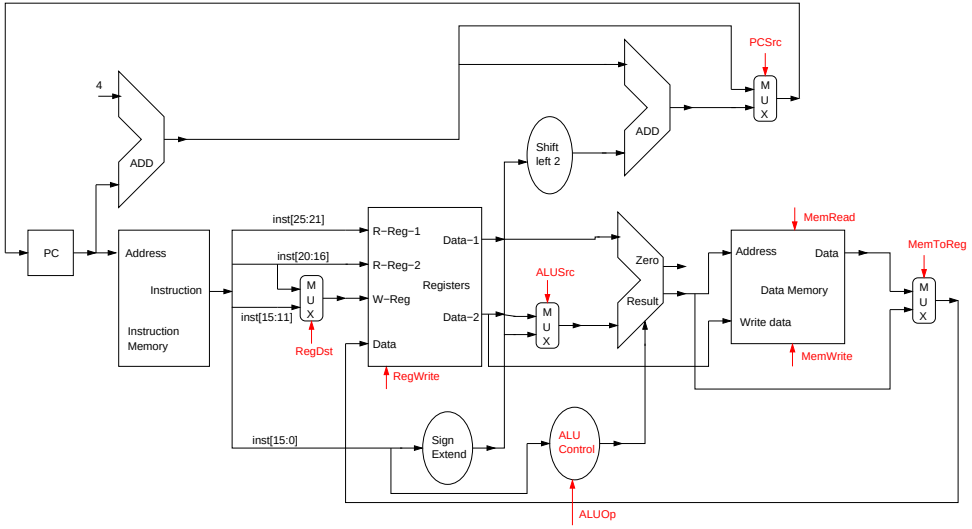


Load or Store instruction



Branch instruction

Full Datapath & Control signals



ALU Control

- Load/Store – ALU performs ADD operation
- Branch – ALU performs SUB operation
- Arithmetic operation – Depends on the instruction

ALU Control lines	Function
0000	AND
0001	OR
0010	ADD
0110	SUB
0111	SLT

ALU Control

- Assume ALUOp derived from opcode

Instruction	ALUOp	Operation	funct	Desired Action	ALU Control input
LOAD	00	load word	XXXXXX	ADD	0010
Store	00	store word	XXXXXX	ADD	0010
Branch equal	01	branch equal	XXXXXX	SUB	0110
R-type	10	add	100000	ADD	0010
R-type	10	subtract	100010	SUB	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	slt	101010	slt	0111

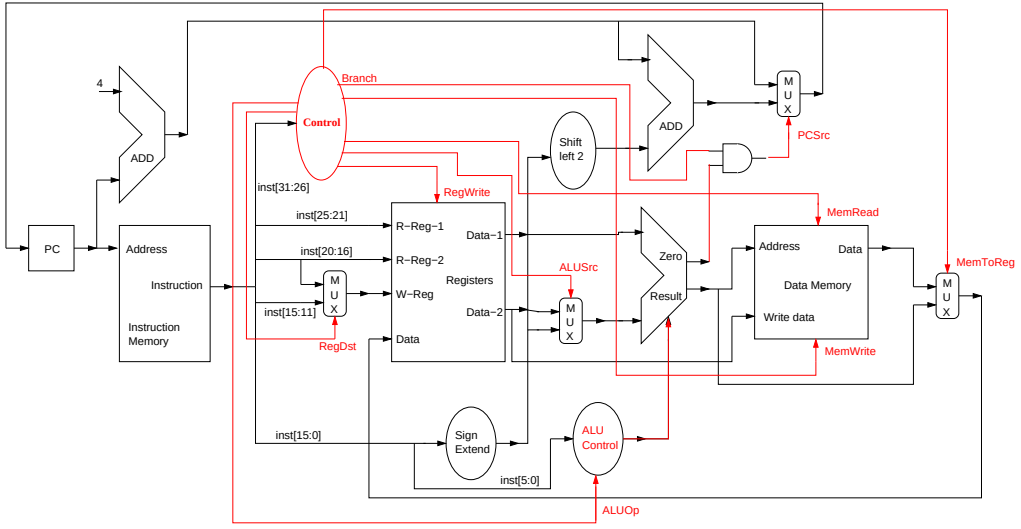
Truth Table for ALU control bits

ALUOp		Funct						
ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

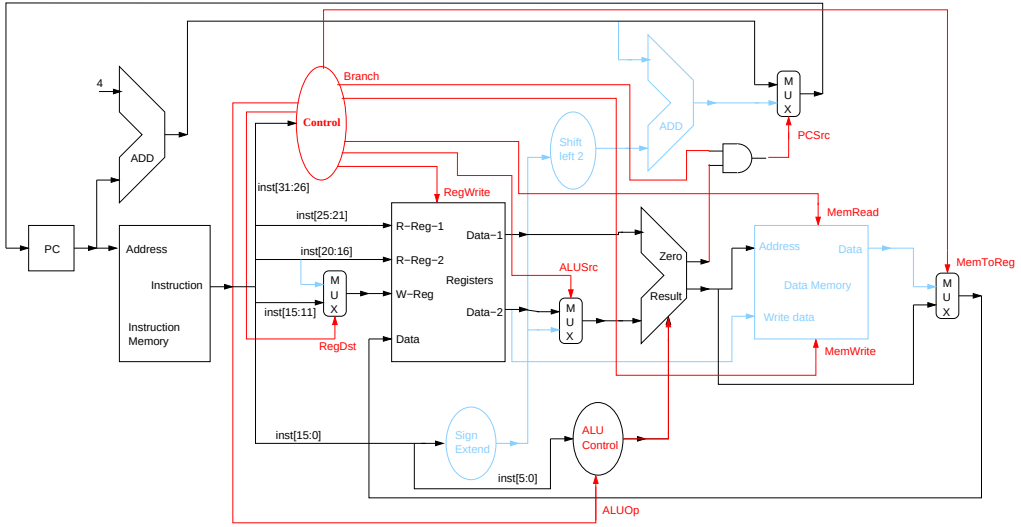
Truth Table for Control lines

Instr	RegDst	ALUsrc	MtoReg	RegW	MemR	MemW	Branch	ALUop1	ALUop2
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

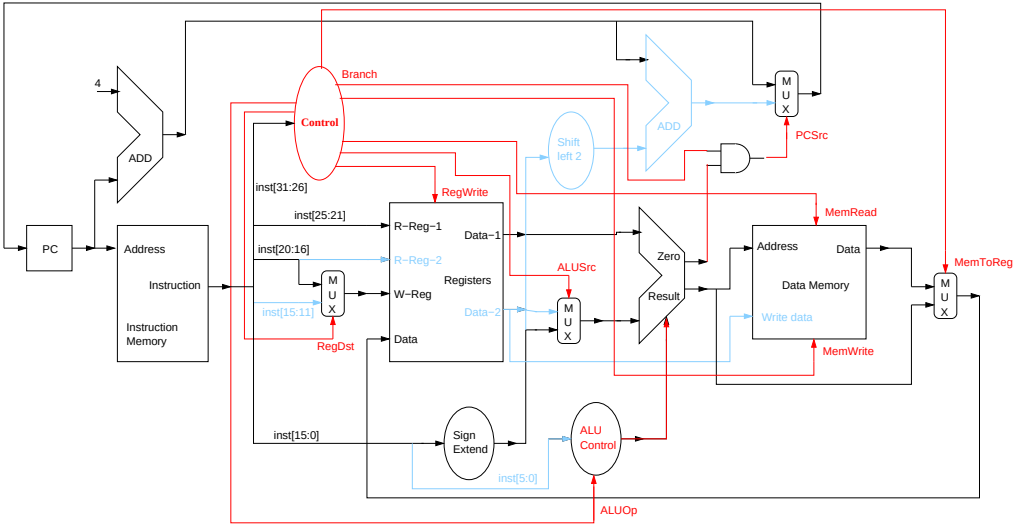
Datapath with Control unit



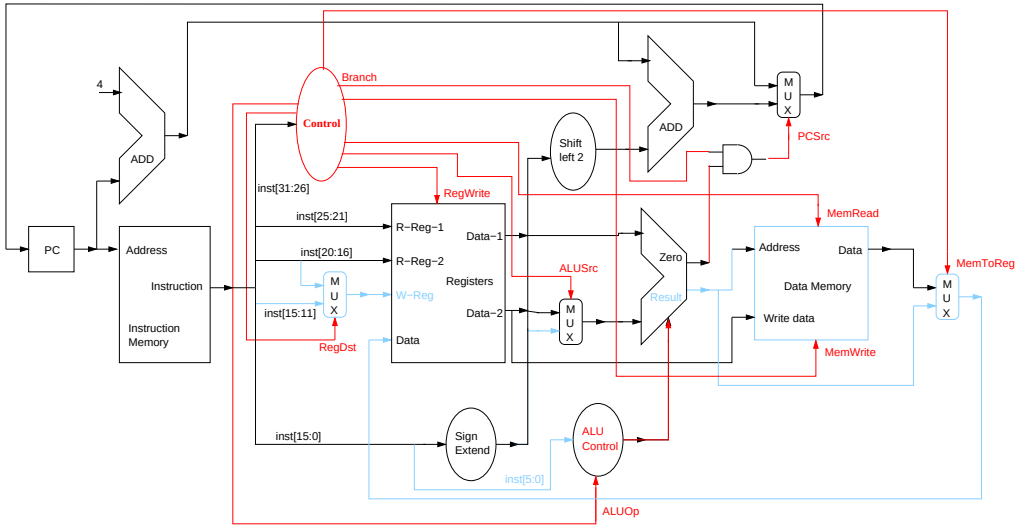
R-type instruction



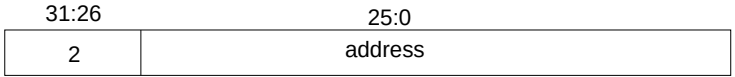
LOAD instruction



Branch if equal

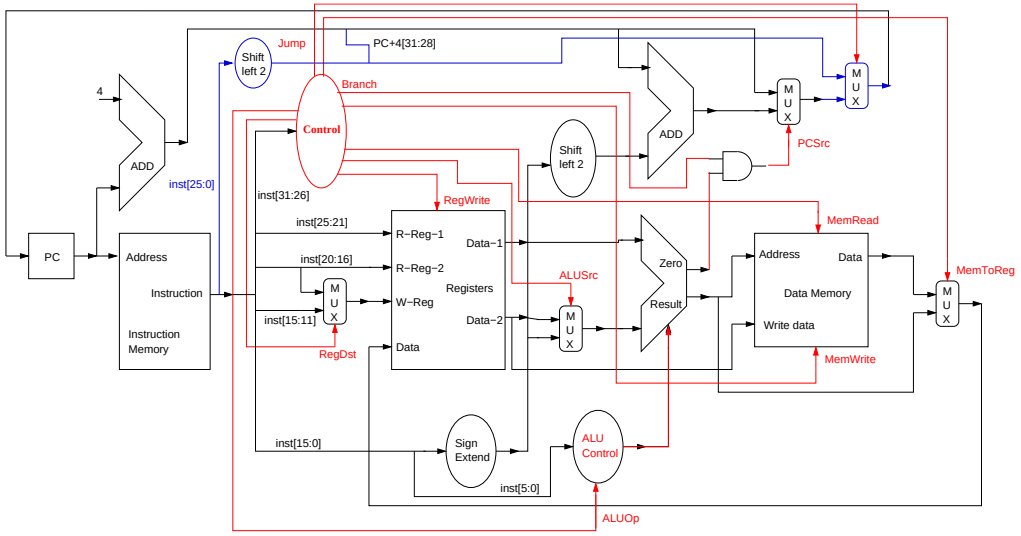


Jump



- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26 bit jump address
  - 00
- Need separate control signal decoded from opcode

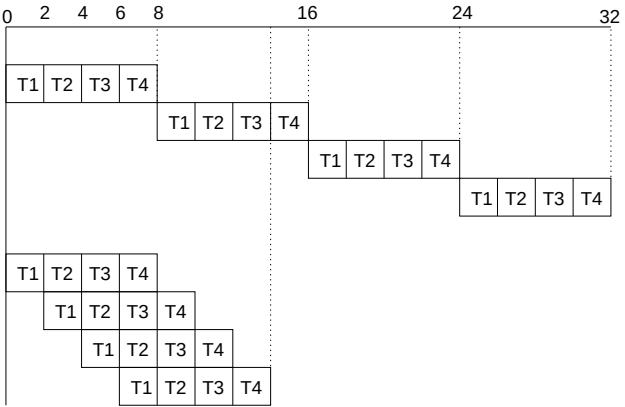
Jump


Jump
PC+4[31:28]
Shift left 2
Branch
4
ADD
Control
inst[25:0]
inst[31:26]
Shift left 2
ADD
M U X
PCSrc
RegWrite
MemRead
inst[25:21]
R−Reg−1
Data−1
PC
Address
Instruction
Instruction Memory
inst[20:16]
R−Reg−2
Registers
W−Reg
Data−2
Data
inst[15:11]
RegDst
ALUSrc
Zero
Result
Address
Data
Data Memory
Write data
MemWrite
MemToReg
inst[15:0]
Sign Extend
ALU Control
inst[5:0]
ALUOp

Performance issue

- Longest delay determines the clock cycle
- Critical path – Load instruction
  - Instruction memory $\rightarrow$ Register files $\rightarrow$ ALU $\rightarrow$ Data memory $\rightarrow$ Register files
- Not feasible to vary clock period for different instruction
- Improve performance by pipelining

Pipelining



- Overlapping execution
- Speed up
  - Four loads: 32/14=2.28
  - Non-stop:
    $8 \times n/(n \times 2 + 6) \approx 4$

MIPS Pipeline

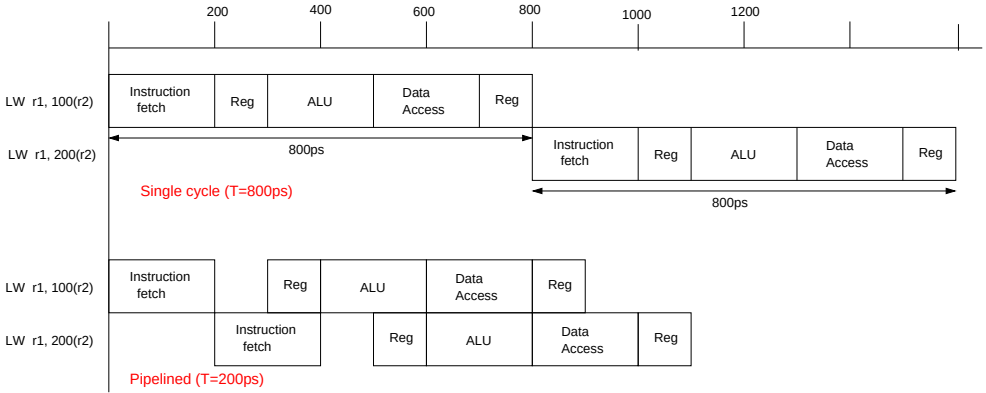
- Five stage pipeline
  - IF : Instruction fetch from memory
  - ID : Instruction decode and register read
  - EX : Execute operation and calculate address
  - MEM : Access memory operand
  - WB : Write result back to memory

Pipeline performance

- Assume time for stages is
  - 100 ps for register read and write
  - 200 ps for rest of the operation

Instruction	Fetch	Reg Read	ALU op	Mem access	Reg write	Total
LW	200	100	200	200	100	800
SW	200	100	200	200		700
R-format	200	100	200		100	600
BEQ	200	100	200			500

Pipelining


200
400
600
800
1000
1200
LW r1, 100(r2)
Instruction fetch
Reg
ALU
Data Access
Reg
LW r1, 200(r2)
800ps
Instruction fetch
Reg
ALU
Data Access
Reg
800ps
Single cycle (T=800ps)
LW r1, 100(r2)
Instruction fetch
Reg
ALU
Data Access
Reg
LW r1, 200(r2)
Instruction fetch
Reg
ALU
Data Access
Reg
Pipelined (T=200ps)

Pipeline speedup

- If all stages are balanced
  - Time between instructions (P) = (Time between instructions (Non-P))/(Number of stages)
- If not balanced, speedup is less
- Speedup due to increased throughput. Time for each instruction remains the same.

Pipeline and ISA design

- MIPS ISA designed for pipelining
- All instructions are 32 bit wide
  - Easier to fetch and decode in one cycle
  - x86: 1 to 17 byte instruction
- Few and regular instruction format
- Load/Store addressing
- Alignment of memory operands

Hazards

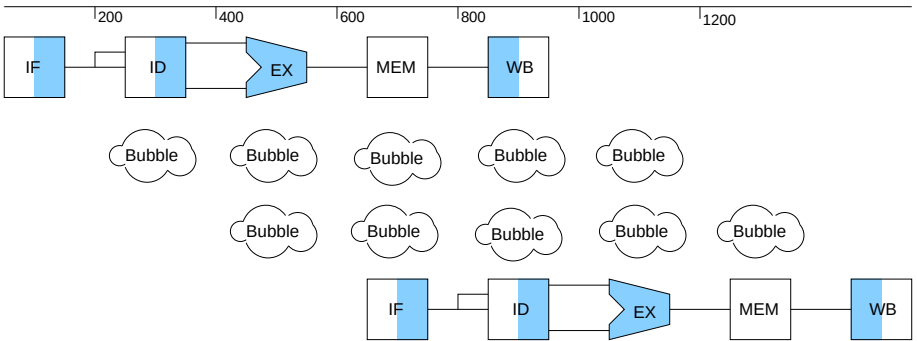
- Situation that prevents starting of next instruction in the next cycle
- Structural hazards
  - Hardware cannot support combination of instructions that are set to execute
  - Required resource is busy
- Data hazards
  - Need to wait for previous instruction to complete its data read/write
    - ⋆ `add $s0, $t0, $t1`
    - ⋆ `sub $t2, $s0, $t3`
- Control hazard
  - Fetched instruction cannot be executed in the proper pipeline clock cycle
  - e.g. BEQ

Structural hazards

- Conflict for use of resource
- In MIPS pipeline with single memory
  - Load/store require memory access
  - Instruction fetch would have to *stall* for that cycle
- Pipelined datapath requires separate data/instrunction memory

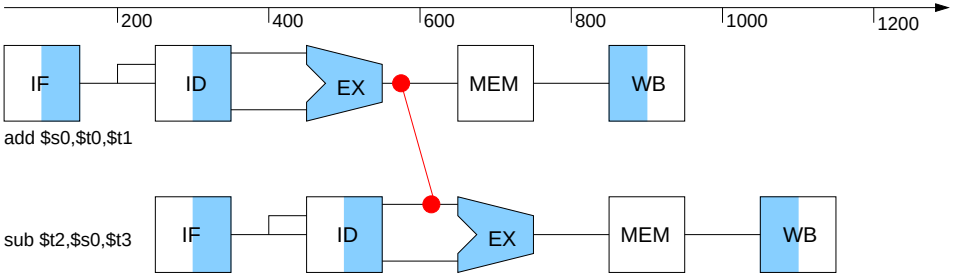
Data hazards

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



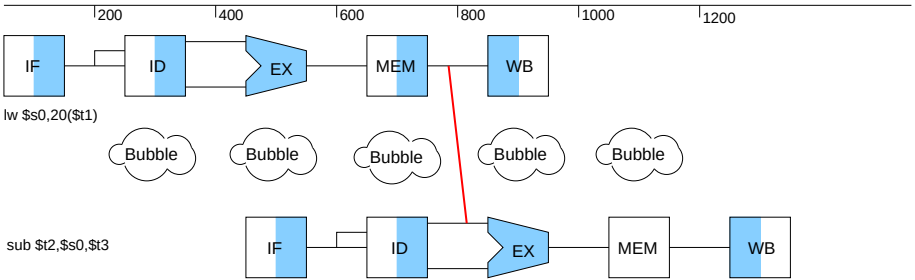
Forwarding

- Use result when it is computed
  - Do not wait for to be stored in reg/mem
  - Requires extra connection in datapath



Load: Data hazard

- Cannot always avoid stalls by forwarding
  - Value not computed when needed
  - Cannot forward backward in time!



Code scheduling to avoid stall

- `a = b + e; c = b + f;`
- Assume all variables are available in memory and are addressable as offset from $t0

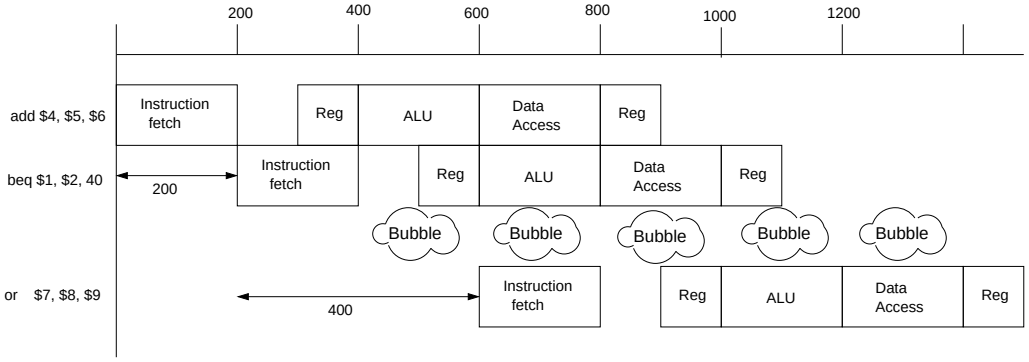
```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Control hazard

- Branch determines the control flow
  - Fetching next instruction depends on the branch outcome
  - Pipeline cannot always fetch correct instruction
- MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

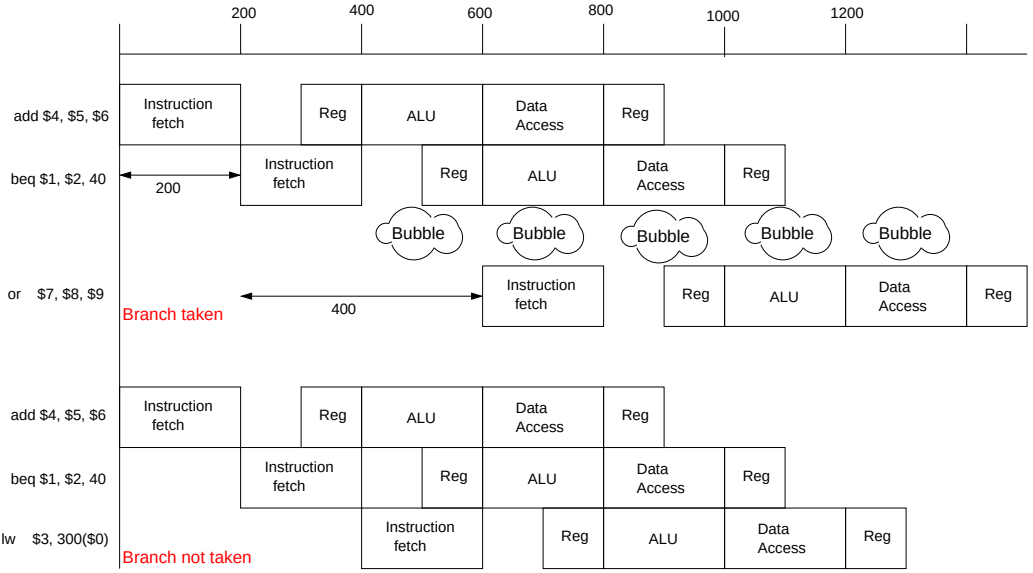
Stall on Branch



Branch prediction

- Larger pipeline cannot determine branch outcome early
  - Stall penalty becomes high
- Predict outcome of branch
  - Stall if outcome is wrong
- MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch with no delay

MIPS branch not taken



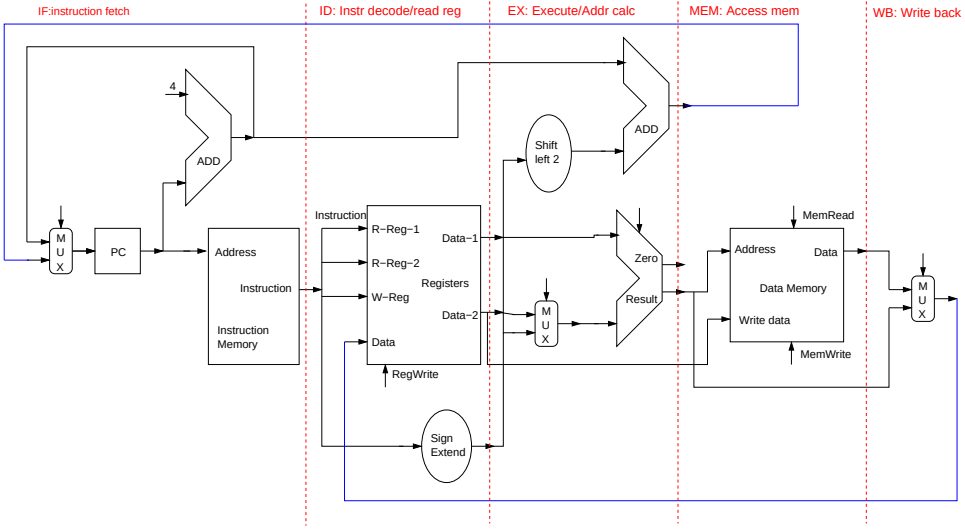
Realistic Branch prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: if-else or loop statements
    - ⋆ Predict backward branches taken
    - ⋆ Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior. Records recent history of each branch.
  - Assume future behavior will continue the trend. If prediction is wrong, stall while re-fetching and update history

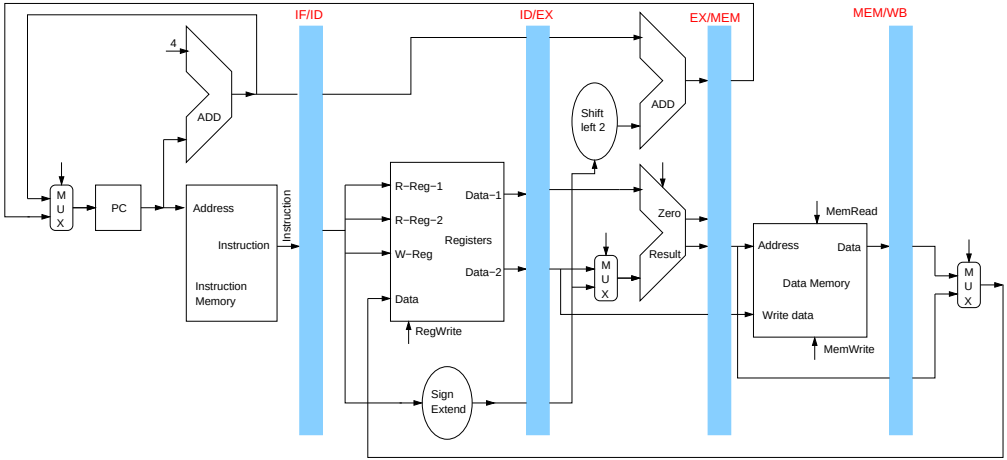
Pipeline summary

- Pipeline increase the performance by increasing instruction throughput
  - Execute multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazard
  - Structural, Data, Control
- Instruction set design affects complexity of pipeline implementation

MIPS Pipeline Datapath



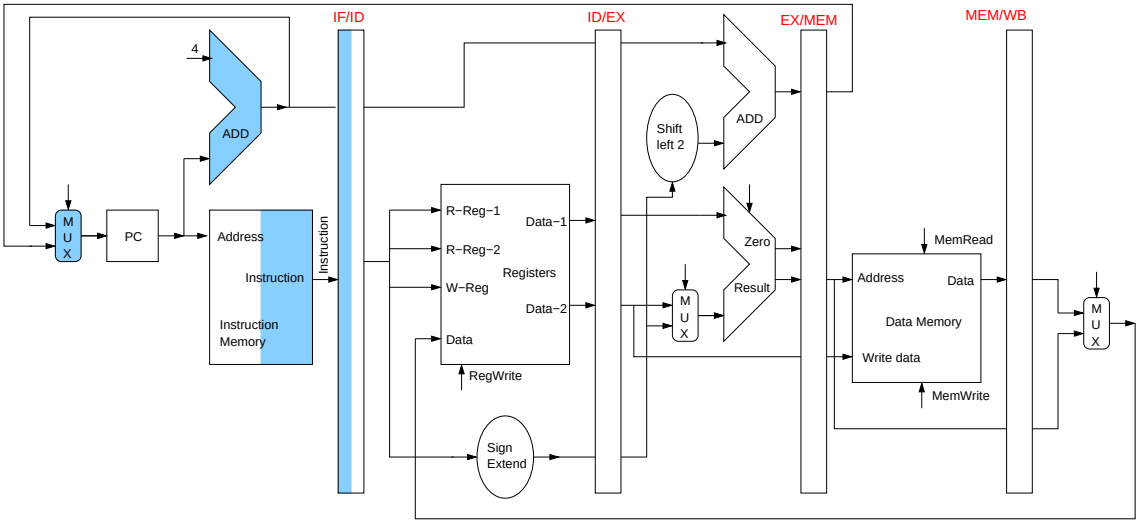
Pipeline Registers



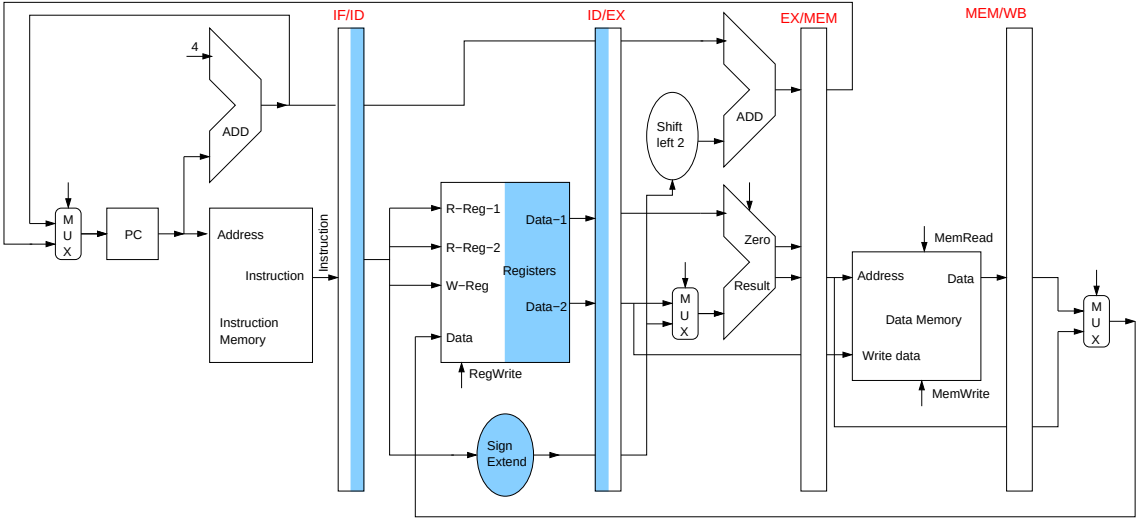
Pipeline operation

- Cycle-by-cycle flow of instruction through the pipelined datapath
  - Single-clock-cycle pipelined diagram
    - ⋆ Shows pipe line usage in single clock cycle
    - ⋆ Highlight resource usage
  - Multi-clock-cycle diagram
    - ⋆ Graph of operation over time

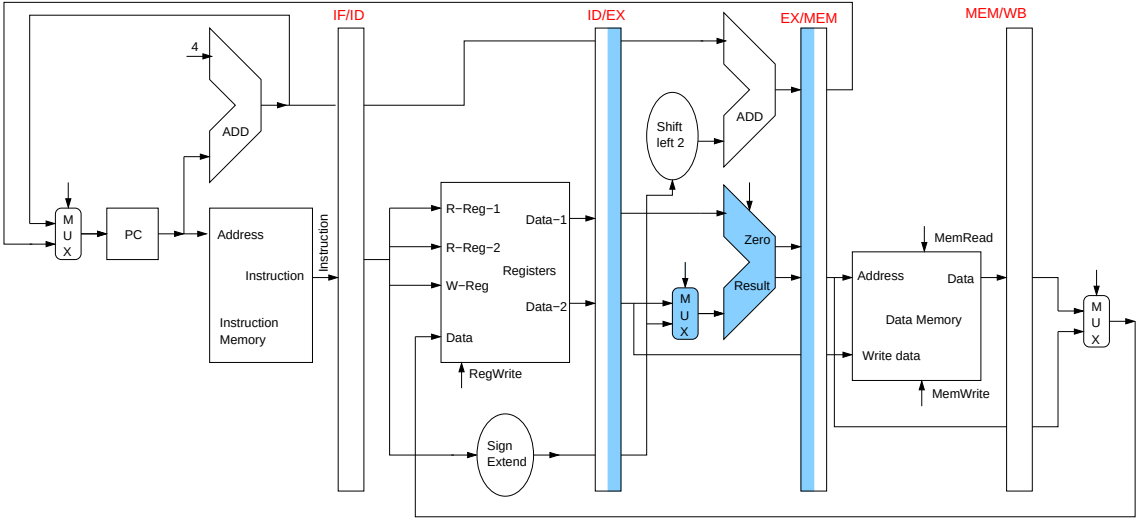
Fetch for LW/SW



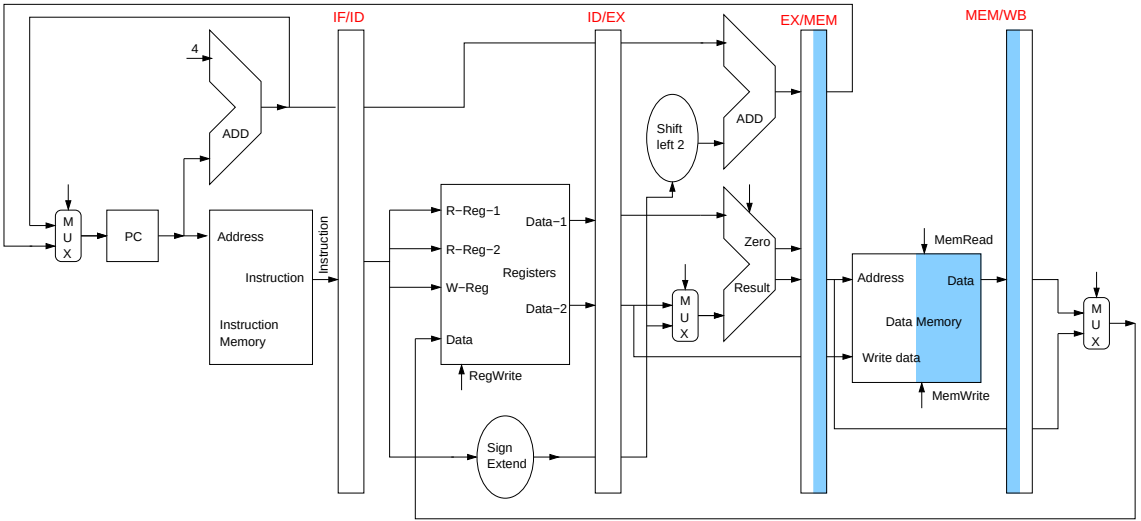
Decode for LW/SW



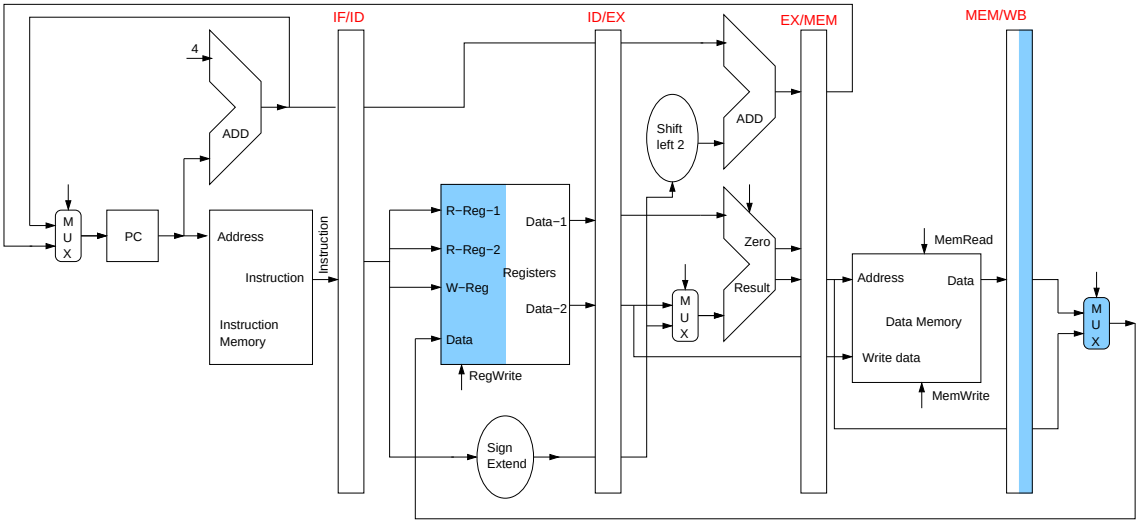
Execute for LW



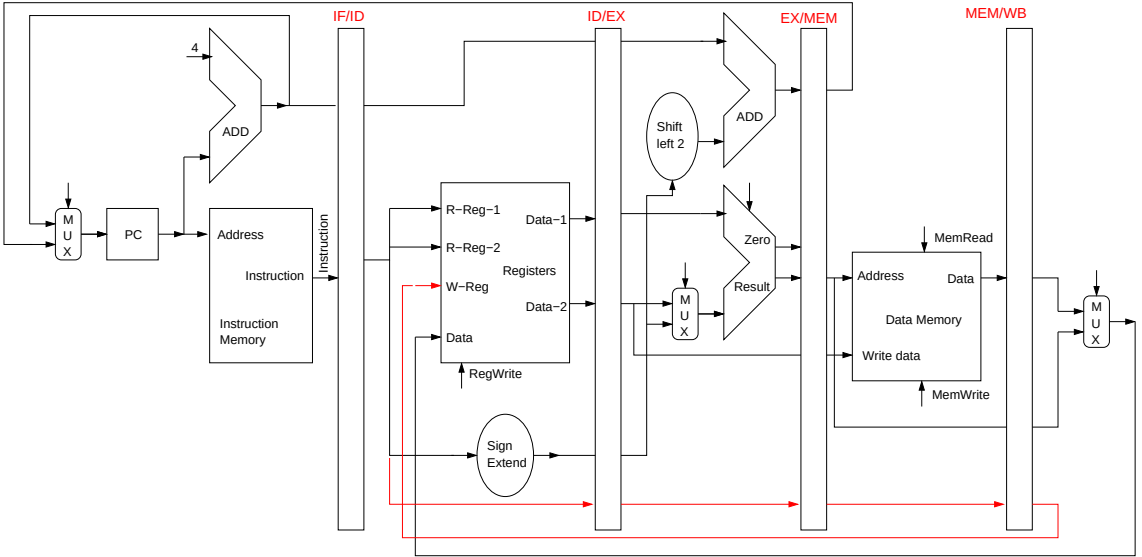
Memory access for LW



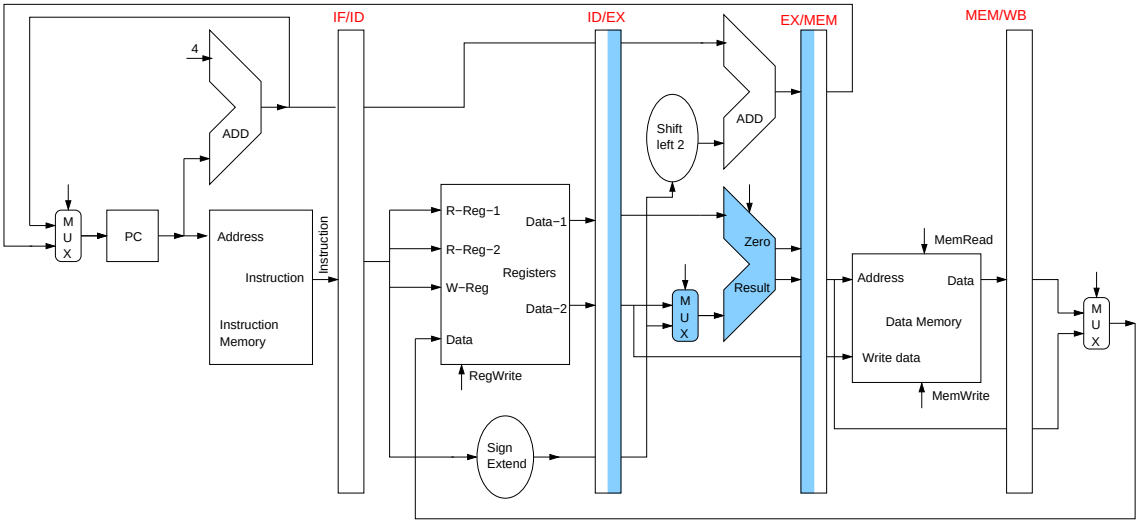
Write-back for LW



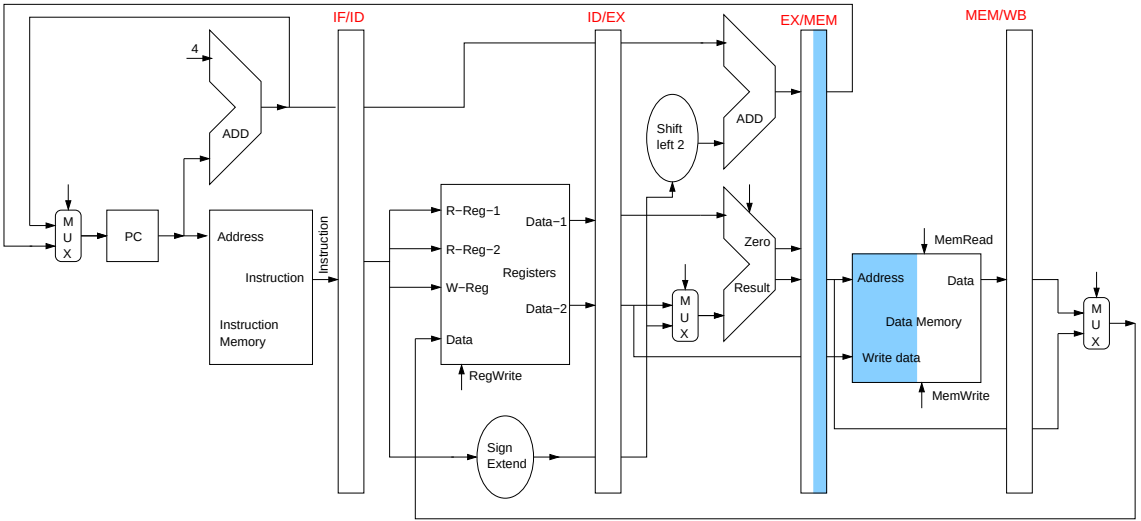
Correct pipeline datapath



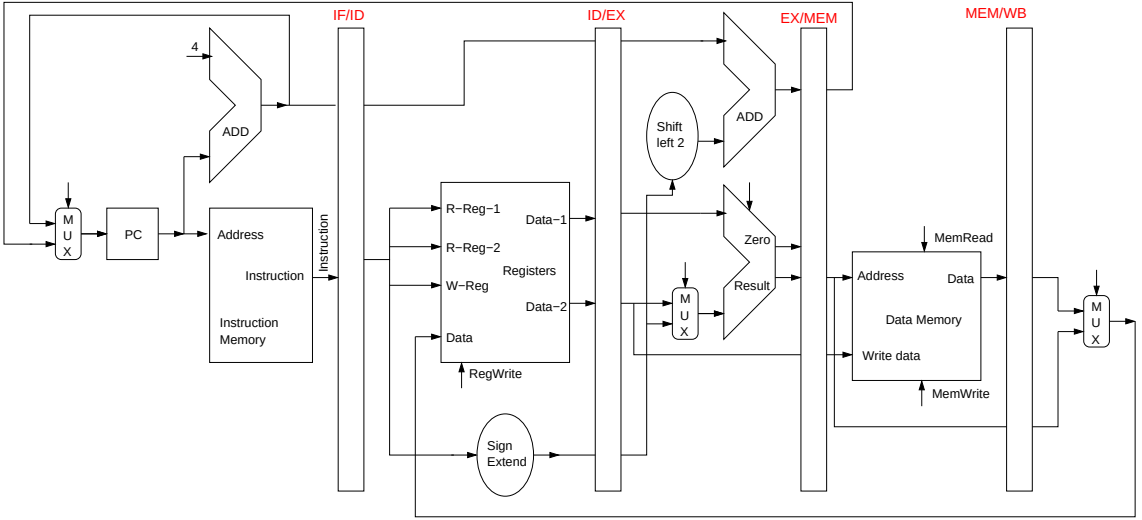
Execute for store



Memory access for store



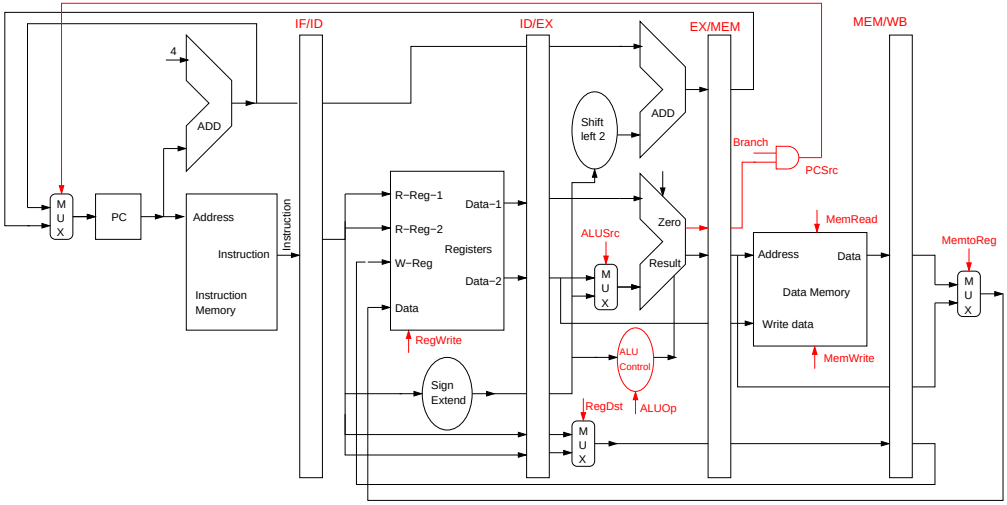
Write-back for store



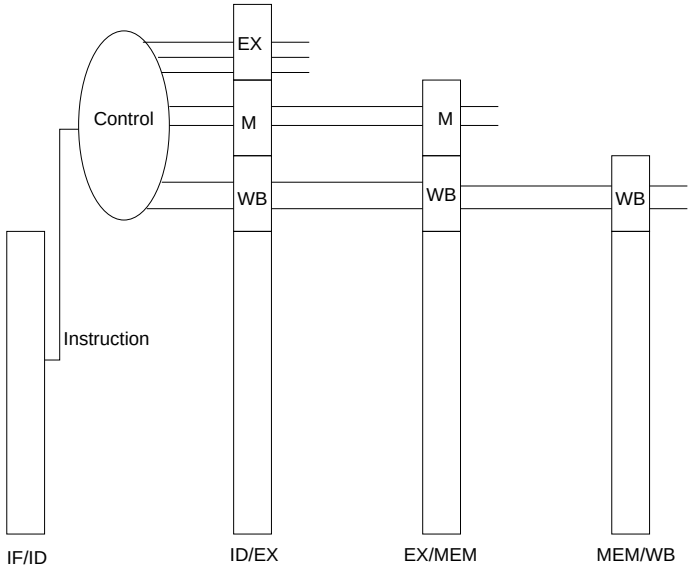
Quiz

- Static branch prediction?
- Dynamic branch prediction?
- A program has two nested loops. Outer loop executes 10 times and inner loop 20 times. Branch instructions are at the end of each loop. What is the accuracy of branch prediction if –
  - Always predict taken
  - Use 1 bit history and it is initialized to 'Taken'.
    - ⋆ Hints: Compute total number of branch instructions executed. Compute the number of mispredictions.
    - ⋆ Open problem: Use 2 bit history

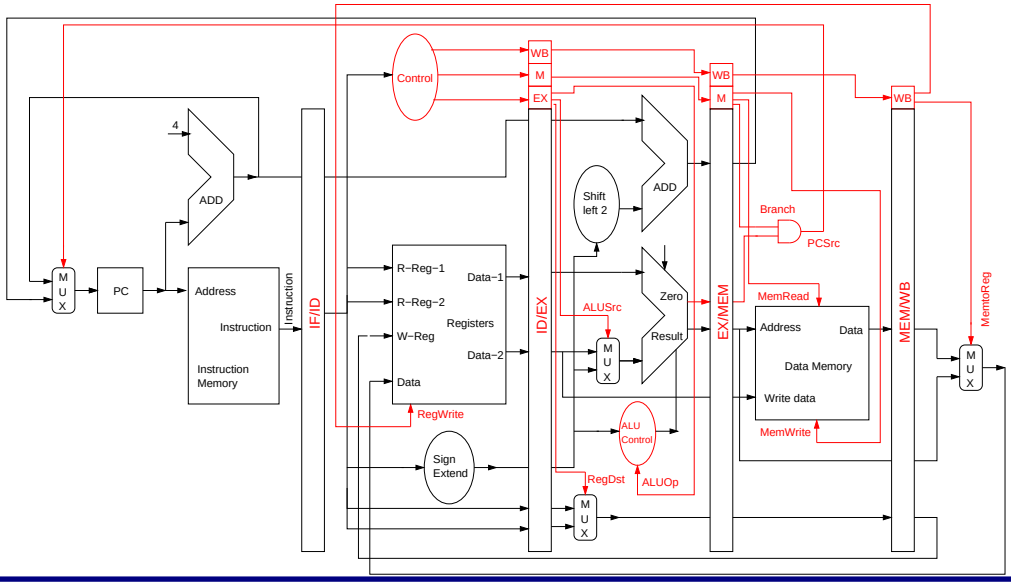
ALU Control


IF/ID
ID/EX
EX/MEM
MEM/WB
4
ADD
M
U
X
PC
Address
Instruction
Instruction
Instruction
Memory
R–Reg–1
R–Reg–2
W–Reg
Data
Registers
Data–1
Data–2
RegWrite
Sign
Extend
Shift
left 2
ADD
ALUSrc
Zero
Result
ALU
Control
RegDst
ALUOp
Branch
PCSrc
MemRead
Address
Data
Data Memory
Write data
MemWrite
MemtoReg

Pipeline Control



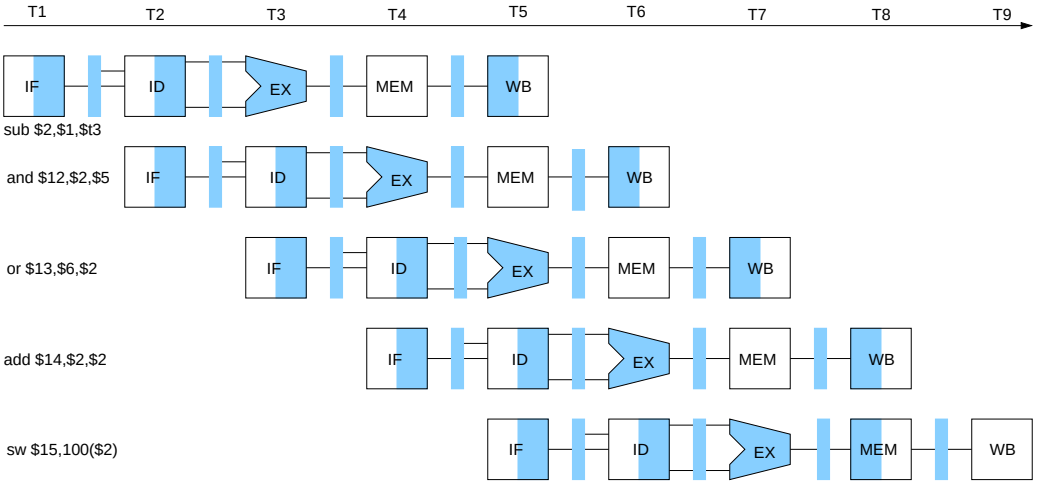
Pipeline Control



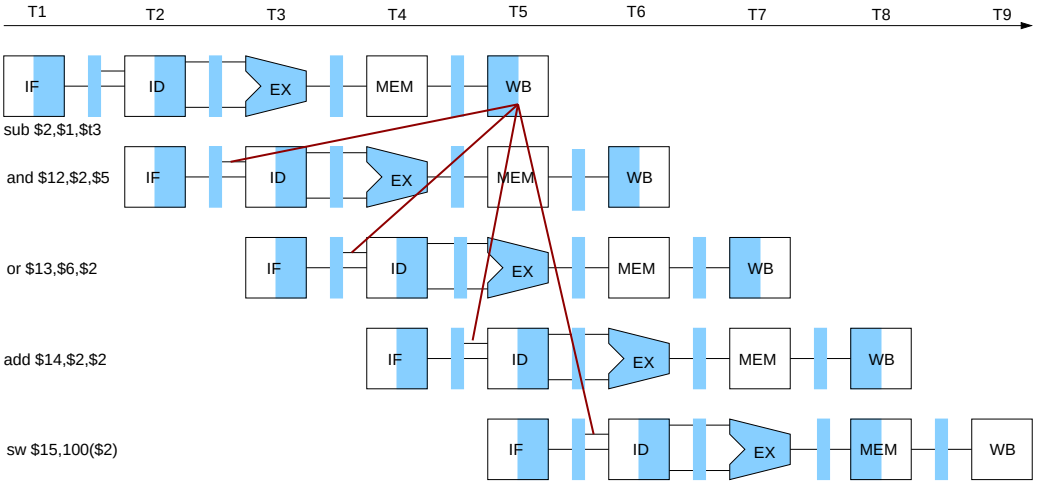
Data hazards in ALU instructions

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

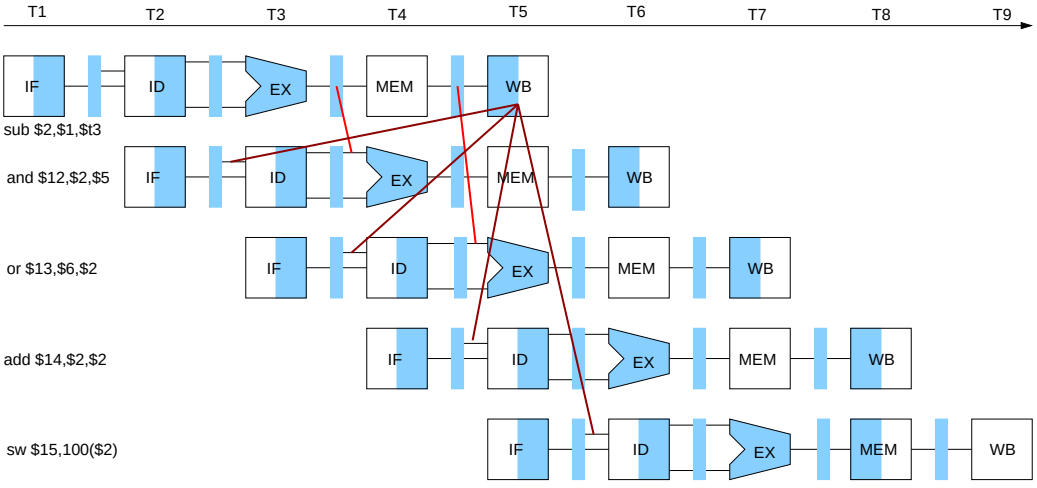
Data hazards



Data hazards



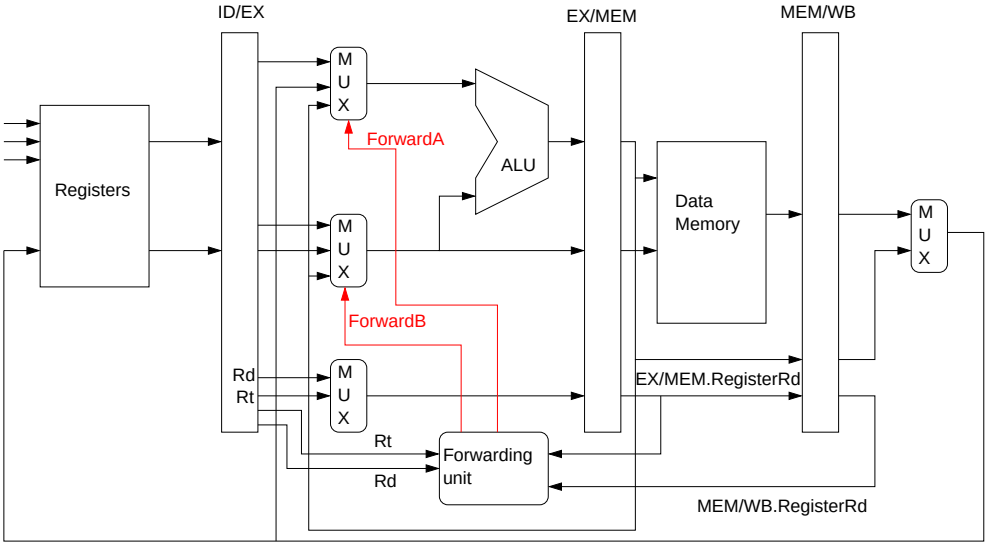
Data hazards



When to forward

- Pass register number along the pipeline
  - ID/EX.RegisterRs – Register number for Rs in ID/EX pipeline register
- Data hazard:
  - EX/MEM.RegisterRd = ID/EX.RegisterRs
  - EX/MEM.RegisterRd = ID/EX.RegisterRt
  - MEM/WB.RegisterRd = ID/EX.RegisterRs
  - MEM/WB.RegisterRd = ID/EX.RegisterRt
- Forwarding instruction will write to a register
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- Destination register is not $0
  - EX/MEM.RegisterRd$\neq$0, MEM/WB.RegisterRd$\neq$0

Forwarding


ID/EX
EX/MEM
MEM/WB
Registers
M
U
X
ALU
Data
Memory
ForwardA
ForwardB
Rd
Rt
EX/MEM.RegisterRd
Forwarding
unit
MEM/WB.RegisterRd

Forwarding conditions

- EX hazard
  - if(EX/MEM.RegWrite & (EX/MEM.RegisterRd $\neq$ 0)
    & (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    $\rightarrow$ ForwardA=10
  - if(EX/MEM.RegWrite & (EX/MEM.RegisterRd $\neq$ 0)
    & (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    $\rightarrow$ ForwardB=10
- MEM hazard
  - if(MEM/WB.RegWrite & (MEM/WB.RegisterRd $\neq$ 0)
    & (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    $\rightarrow$ ForwardA=01
  - if(MEM/WB.RegWrite & (MEM/WB.RegisterRd $\neq$ 0)
    & (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    $\rightarrow$ ForwardB=01

Double Data Hazard

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

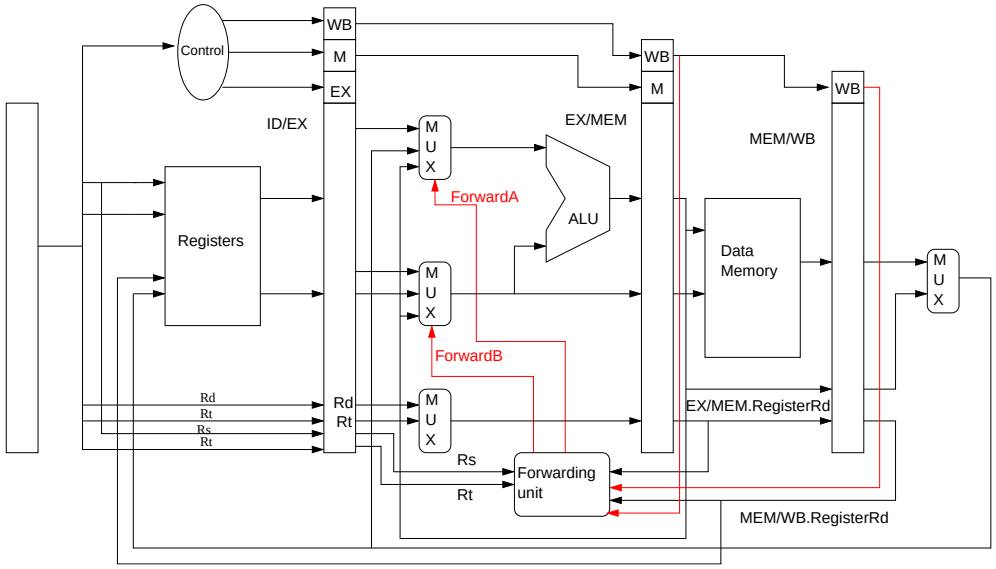
- Both hazards occur. Want to use the most recent one
- Need to revise MEM condition. Forward only if EX hazard condition is not true.

Revise Forwarding conditions

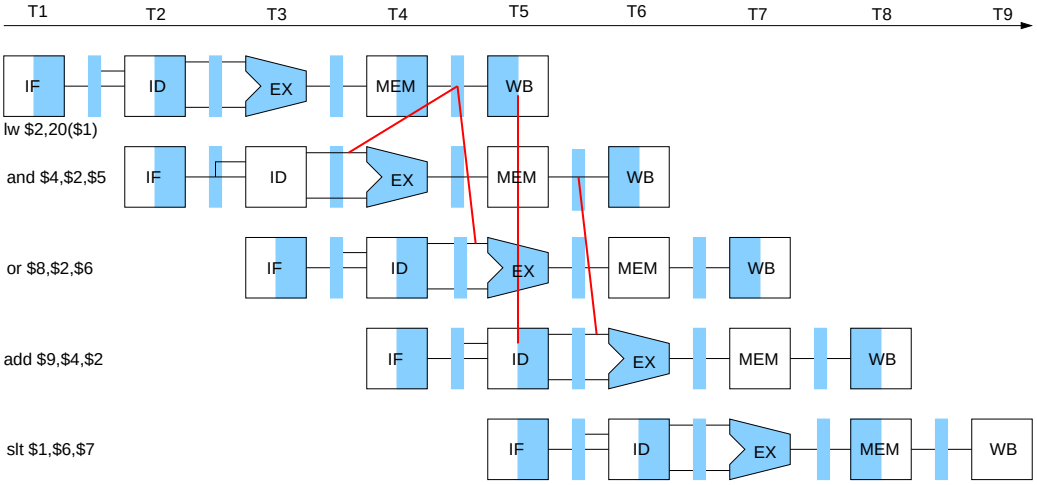
- MEM hazard
  - if(MEM/WB.RegWrite & (MEM/WB.RegisterRd $\neq$ 0)
    & !(EX/MEM.RegWrite & EX/MEM.RegisterRd $\neq$ 0) & (EX/MEM.RegisterRd $\neq$ ID/EX.RegisterRs)
    & (MEM/WB.RegisterRd = EX/MEM.RegisterRs))
    $\rightarrow$ ForwardA=01

  - if(MEM/WB.RegWrite & (MEM/WB.RegisterRd $\neq$ 0)
    & !(EX/MEM.RegWrite & EX/MEM.RegisterRd $\neq$ 0) & (EX/MEM.RegisterRd $\neq$ ID/EX.RegisterRt)
    & (MEM/WB.RegisterRd = EX/MEM.RegisterRt))
    $\rightarrow$ ForwardB=01

Forwarding


Control
WB
M
EX
ID/EX
WB
M
EX/MEM
WB
MEM/WB
M
U
X
ForwardA
ALU
Registers
Data
Memory
M
U
X
ForwardB
Rd
Rt
Rs
Rt
Rd
Rt
M
U
X
EX/MEM.RegisterRd
Rs
Rt
Forwarding
unit
MEM/WB.RegisterRd

Load use data hazard



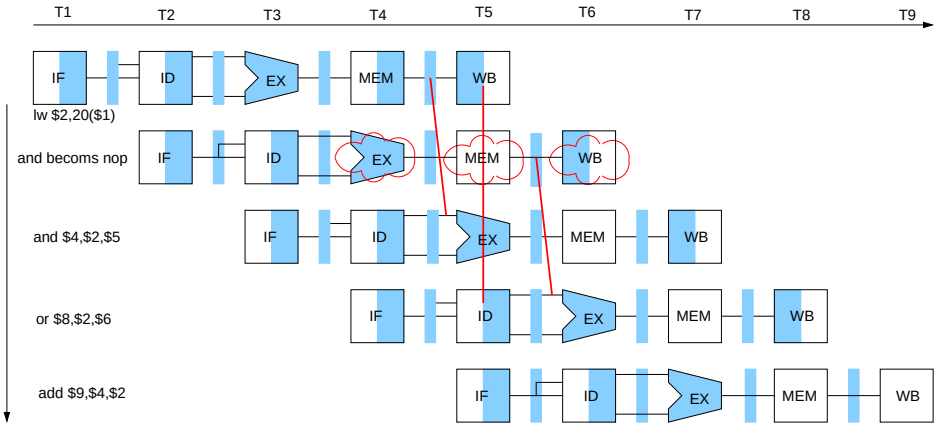
Load Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage is given by IF/ID.RegisterRs and IF/ID.RegisterRt
- Load use hazard:
  - if(ID/EX.MemRead &
    ((ID/EX.RegisterRt = IF/ID.RegisterRt) || (ID/EX.RegisterRt = IF/ID.RegisterRs)))
    stall the pipeline, insert bubble

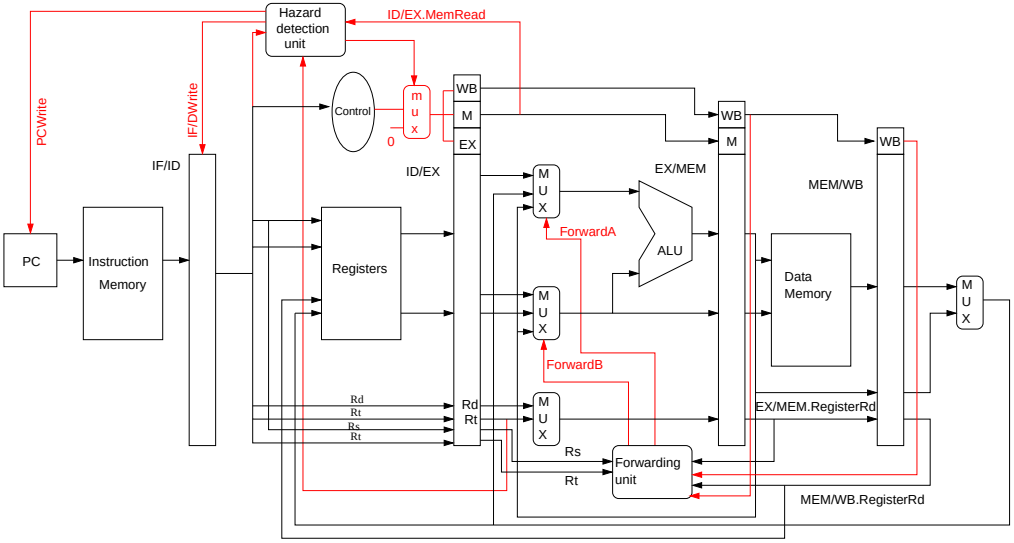
How to stall the pipeline

- Force control values in ID/EX register to 0
- EX, MEM and WB will do nop
- 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

Bubble in the pipeline



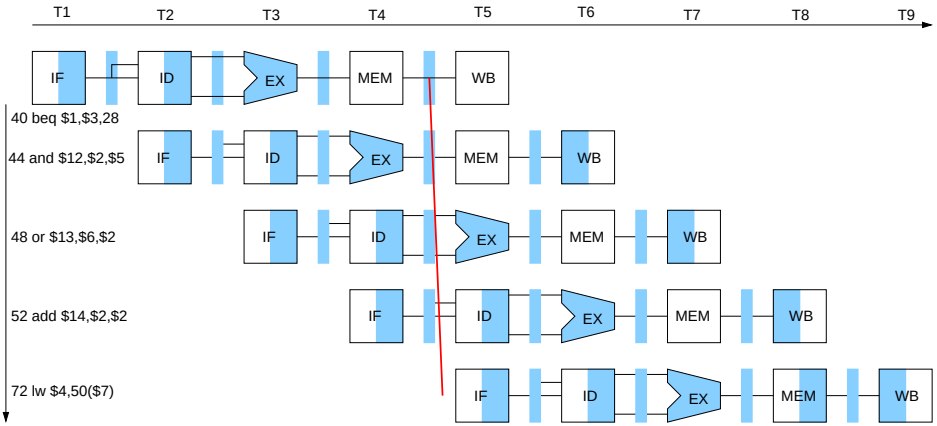
Hazard detection unit



Pipeline: Stall & Performance

- Stalls reduce performance but it is required to get the correct results
- Compiler can arrange code to avoid hazards and stalls. Compiler need to have the knowledge of pipeline structure

Branch Hazard

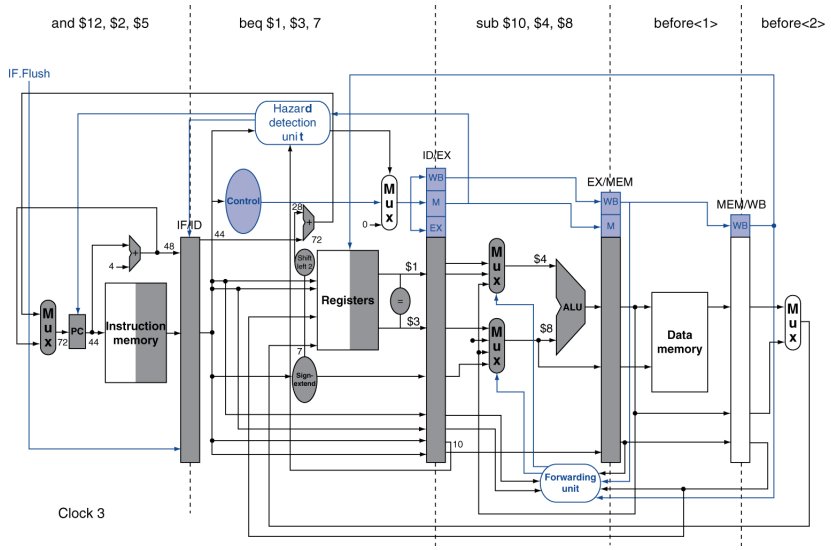


Reducing Branch delay

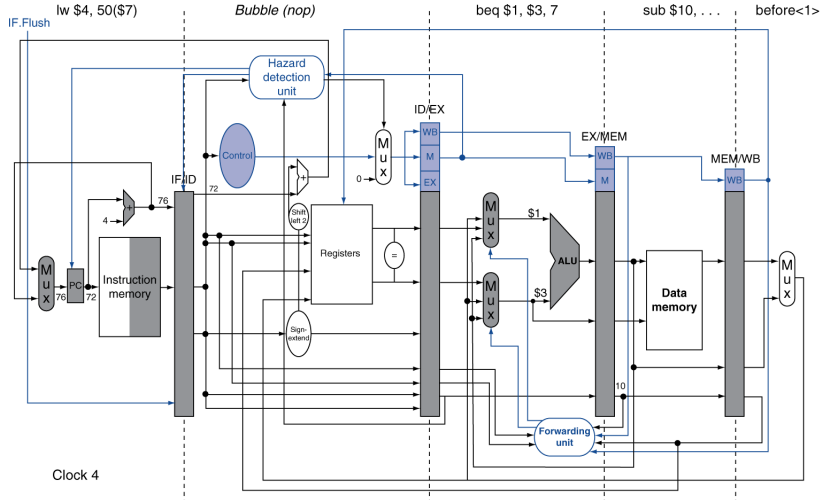
- Hardware required to determine outcome in ID stage
  - Target address adder
  - Register comparator
- Example:

```
36:  sub $10, $4, $8
40:  beq $1,  $3, 7
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)
```

Branch Hazard

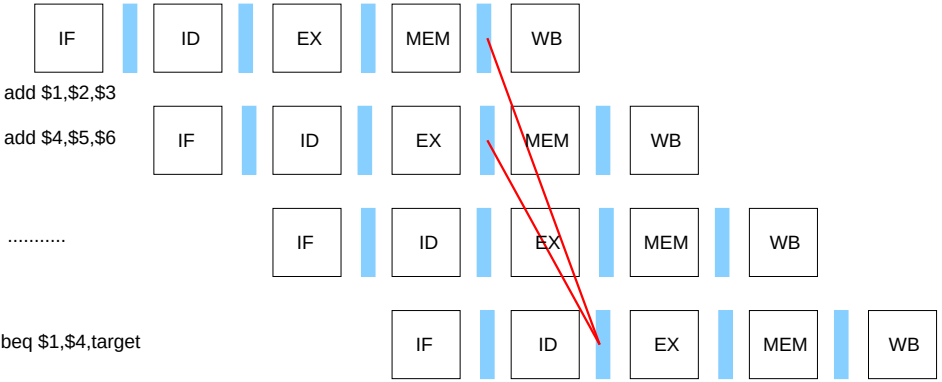


Branch Hazard



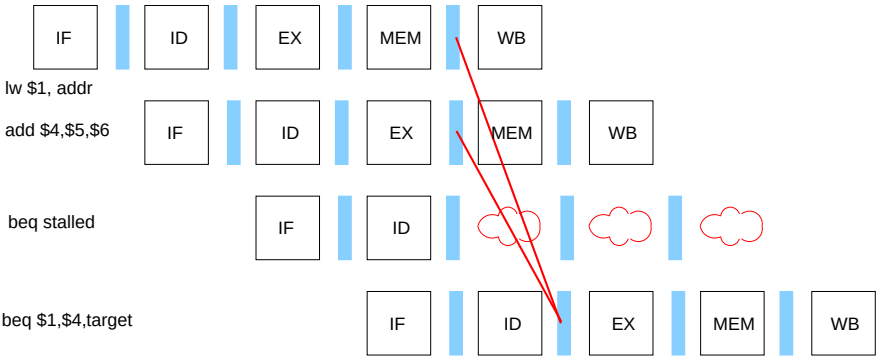
Data Hazard for Branch

- A comparison register is a destination of 2nd or 3rd preceding ALU instruction
- Can be resolved using forwarding



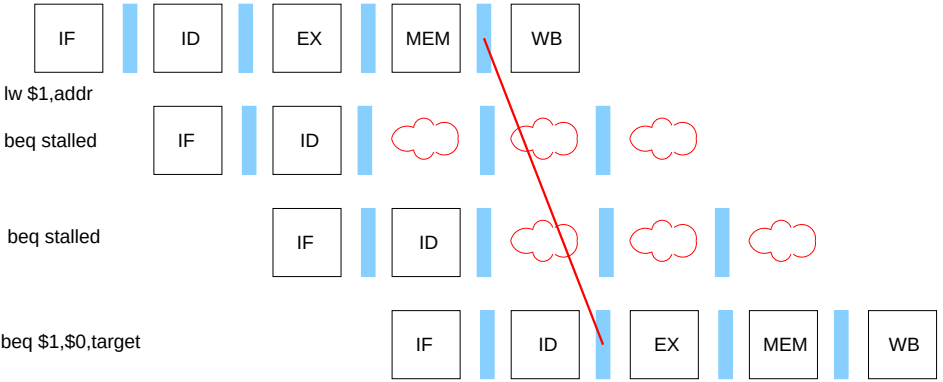
Data Hazard for Branch

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



Data Hazard for Branch

- A comparison register is a destination of preceding LOAD instruction
- Need 2 stall cycles



Dynamic Branch Prediction

- Branch penalty is significant in deeper pipeline
- Use dynamic prediction
  - Branch prediction buffer
  - Indexed by recent branch instruction address
  - Stores outcome (taken/not-taken)
  - To execute a branch
    - ⋆ Check table expect the same outcome
    - ⋆ Start fetching from fall-through or target
    - ⋆ If wrong, flush pipeline and flip prediction

1-bit predictor

```
outer:  ....
        ....
inner:  ....
        ....
        beq inner
        ....
        beq outer
```

- Inner loop branch mispredicts twice
  - Mispredict as 'taken' on last iteration of the inner loop
  - Mispredict as 'not-taken' on first iteration of inner loop next time

2-bit predictor

- Change prediction on two successive mispredictions
- Solve quiz problem!!

Branch target calculation

- Even with the predictor, target address need to be determined
  - 1 cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction is fetched
  - If branch predicted is taken and its a hit, then target address can be fetched immediately

Quiz

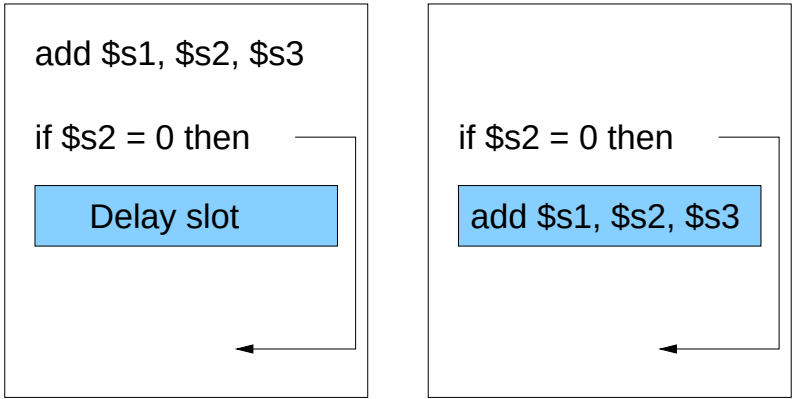
```
I1:  lw  $1,40($2)
I2:  add $2,$3,$3
I3:  add $1,$1,$2
I4:  sw  $1,20($2)
```

- Find all data dependencies – `RAW, WAR, WAW`
- Find all hazards for five stage pipeline with and without forwarding.

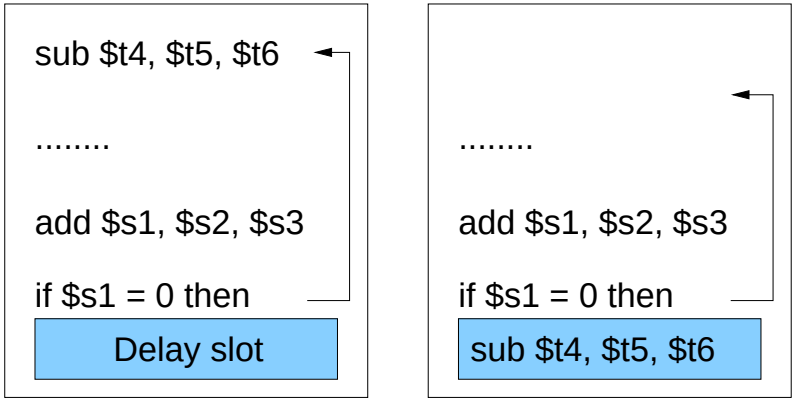
Pipeline Conclusion

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control

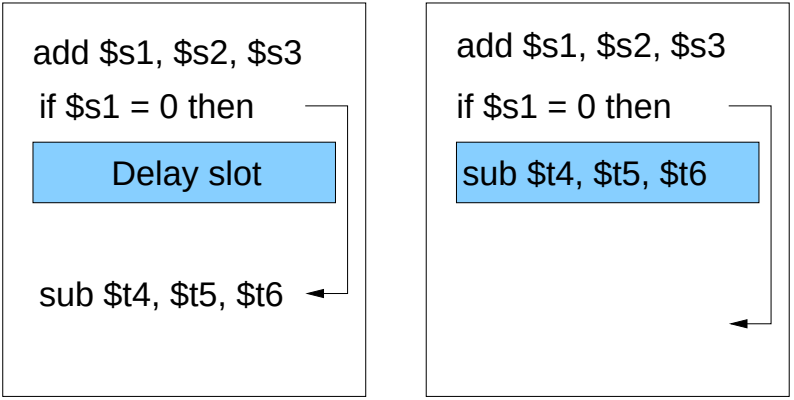
Branch Delay Slot: Example-1



Branch Delay Slot: Example-2



Branch Delay Slot: Example-3



Exception and Interrupt

- Unexpected event requires change in flow of control
- Exceptions: Arises within the cpu eg. undefined opcode, overflow, syscall, ...
- Interrupt: From external I/O controller
- Dealing with them without sacrificing performance is hard

Handling exception

- In MIPS, exceptions are managed by System Control Coprocessor (CPO)
- Save PC of offending (interrupted) instruction. In MIPS, EPC (Exception Program Counter)
- Save indication of the problem
- Jump to handler at 8000 0180

An alternate mechanism

- Vectored interrupt: Handler address is determined by the cause of the problem
- Example:
  - Undefined opcode C000 0080
  - Overflow C000 0180
  - C000 0280
- Instructions are either
  - Deal with the interrupt or
  - Jump to real handler

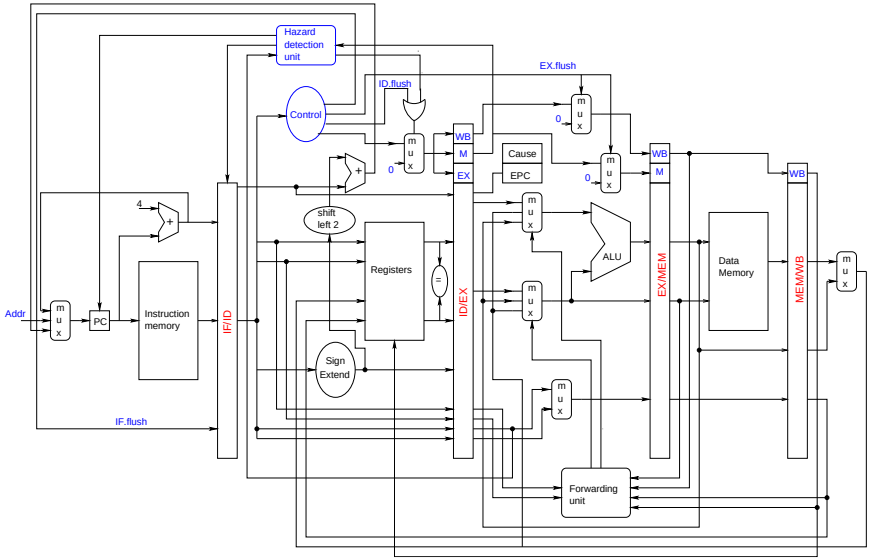
Handler action

- 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

Exception in Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage – `add $1,$1,$2`
  - Prevent $1 from being corrupted
  - Complete previous instruction
  - Flush 'add' and subsequent instruction
  - Set Cause and EPC register
  - Transfer control to handler
- Similar to mispredict. Use much of the same hardware.

Pipeline with exception



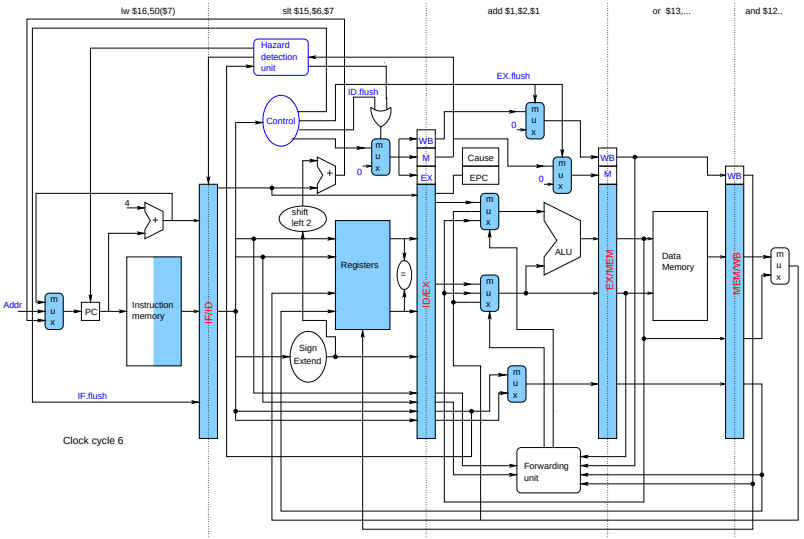
Exception properties

- Restartable exception
  - Pipeline can flush the instruction
  - Handler executes, then return to the instruction
    - ⋆ Refetched and executed from scratch
- PC saved in EPC register
  - Identify causing instruction
  - Actually PC+4 is saved
    - ⋆ Handler must take care of it

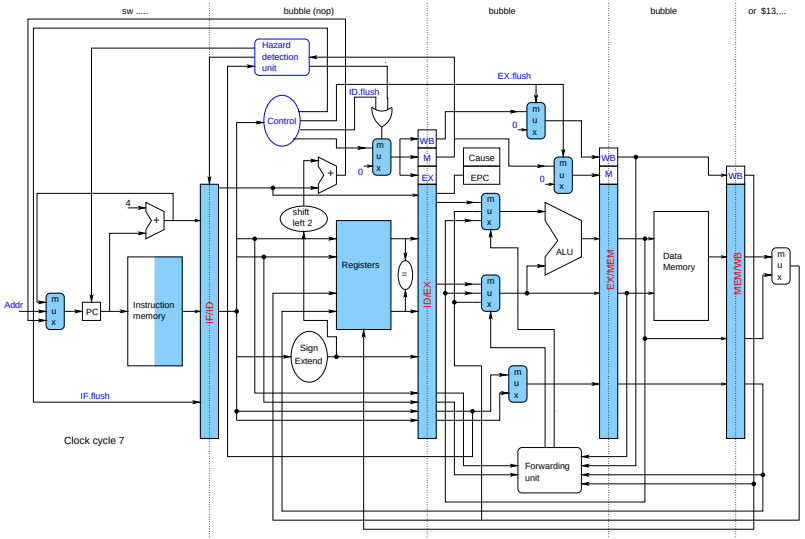
Exception example

```
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
64   lw  $16, 50($7)
......
8000 0180     sw .....
8000 0184     sw .....
```

Exception example



Exception example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exception at once
- Simple approach: Serve exception from the earliest instruction
  - Flush subsequent instructions
  - "Precise" exception
- Complex pipeline
  - Multiple instructions are issued per cycle
  - Out-of-order completion
  - Maintaining precise exception is difficult

Imprecise Exceptions

- Stop pipeline and save state
  - Including exception causes
- Let the handler work out
  - Which instructions had exceptions
  - Which to complete and to flush
- Simplifies hardware but more complex handler software

Quiz

```
I1:  lw  $1,40($6)
I2:  add $2,$3,$1
I3:  add $1,$6,$4
I4:  sw  $2,20($4)
I5:  and $1,$1,$4
```

- If there is no forwarding, no hazard detection, insert nops to ensure the correct operation
- Repeat the same but now use nops when a hazard can be avoided by changing or rearranging these instructions. You can use $7 to store temporary values in your modified code.

Instruction level parallelism (ILP)

- Executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - ⋆ Less work per stage $\Rightarrow$ shorter clock
  - Multiple issue
    - ⋆ Replicate pipeline stages $\Rightarrow$ multiple pipeline
    - ⋆ Start multiple instructions per clock cycle
    - ⋆ CPI $<$ 1, so use IPC (Instructions Per Clock cycle)
    - ⋆ 4GHz 4 way multiple issue : CPI - 0.25, IPC - 4

Multiple Issue

- Static multiple issue
  - Compiler group instructions that can be issued together
  - Packages them into issue slot
  - Compiler detects and avoid hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instruction to issue each cycle
  - Compiler can help by reordering the instructions
  - CPU resolves hazards using advance techniques at runtime

Speculation

- Guess what to do with instruction
  - Start operation as soon as possible
  - Check whether guess is right or not
    - ⋆ If right, continue with the instruction
    - ⋆ If worng, roll back and do the right thing
- These problems are valid for both static and dynamic multiple issue
- Examples:
  - Speculate on branch outcome: Roll back if different path is taken
  - Speculate on load: Roll back if location is updated

Software/Hardware Speculation

- Software
  - Compiler can reorder instructions, eg. move load instruction before branch
  - Insert additional instruction to check accuracy
  - Provides a fixed up instruction
- Hardware:
  - Buffers speculative results until it determines they are needed
  - Incorrect speculation requires to flush the buffers anf reexecute correct instruction sequence.

Speculation & Exception

- Example: Illegal access of address when speculation is incorrect
- More complicated situation: Illegal access of address when speculation is *correct*
- Static speculation:
  - Add ISA support for deferring exception
- Dynamic speculation:
  - Buffer exceptions until the instruction completion
- Performance improves on correct speculation

Static multiple issue

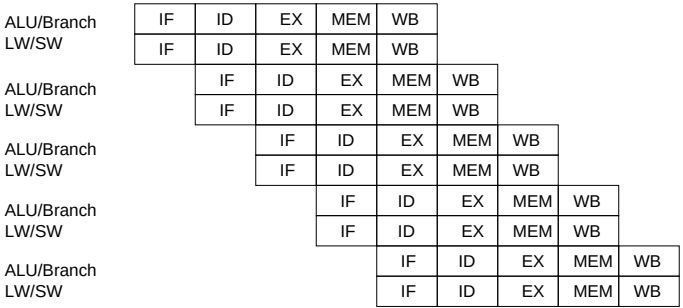
- Compiler groups instructions into issue packets
  - Group of instructions that can be issued in a single clock cycle
  - Determined by pipeline resource usage
- Think of an issue packet as very long instruction
  - Specify multiple concurrent operation
  - Very Long Instruction Word (VLIW)

Scheduling static multiple issue

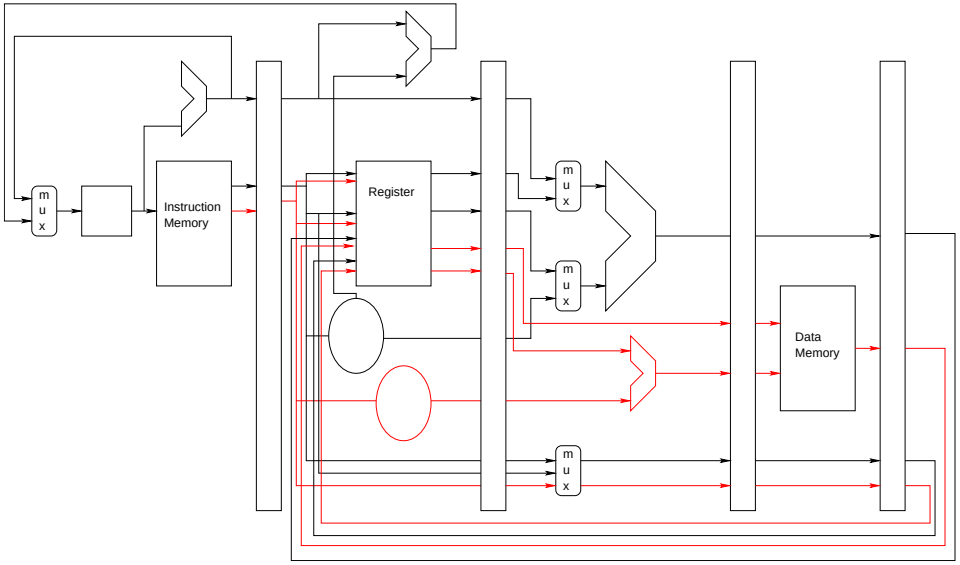
- Compiler must remove all/some hazards
  - Reorder instructions into issue packets
  - No dependency within a packet
  - Possibly some dependency between packets
  - Pad with nop if necessary

MIPS with static dual issue

- Two issue packets
  - One ALU/Branch instruction
  - One load/store instruction
  - 64 bit aligned



MIPS with static dual issue



Hazards in Dual issue MIPS

- More instructions are executed in parallel
- EX data hazard
  - Forwarding avoided stall in single issue
  - Now cannot use ALU results in load/store in same packet

```
add $t0, $s0, $s1
load $s2, 0($t0)
```

    - ⋆ Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle latency, but now two instructions

Scheduling example

```
loop:  lw $t0, 0($1)
       addu $t0, $t0, $2
       sw $t0, 0($1)
       addi $s1, $s1, -4
       bne $s1, $0, loop
```

```
loop:  nop;                    lw $t0, 0($1)
       addi $s1, $s1, -4;       nop;
       addu $t0, $t0, $2;       nop;
       bne $s1, $0, loop;       sw $t0, 4($1)
```

Loop unrolling

- Replicate loop body to expose more parallelism
- Reduces loop control overhead
- Use different registers per replication
  - Called register renaming
  - Avoid loop carried anti-dependencies
    - ⋆ Store followed by a load of the same register
    - ⋆ Known as name dependance

Loop unrolling example

	ALU/Branch	Load/Store	Cycle
Loop:	addi $1, $1, -16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $0, Loop	sw $t3, 4($s1)	8

Dynamic multiple issue

- Superscalar processor
- CPU decide whether to issue 0,1,3,... in each cycle
  - Avoiding structural and data hazard
  - Avoid loop carried anti-dependencies
- Avoids the need for compiler scheduling
  - Still compiler can help
  - Code semantics ensured by CPU

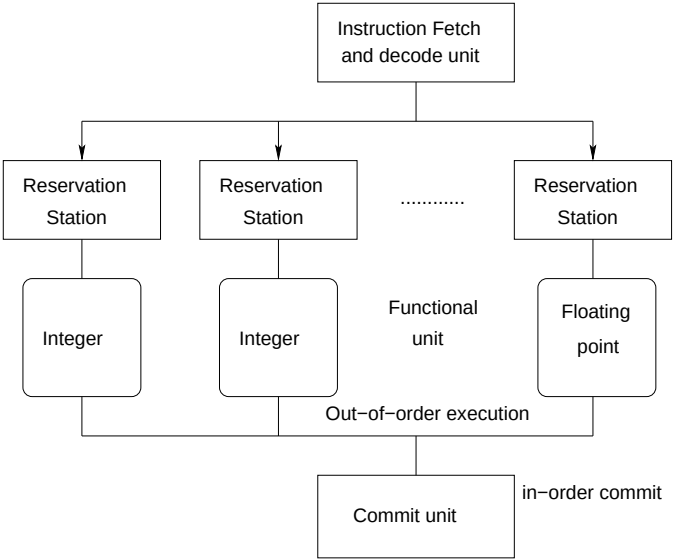
Dynamic pipeline scheduling

- Allow the CPU to execute instructions out-of-order to avoid stall but commit result to register in order
- Example:

```
lw   $t0, 20($s2)
addu $t1, $t0, $t2
sub  $s4, $s4, $t3
slti $t5, $s4, 20
```

  - Can start sub while addu is waiting for lw to complete

Dynamically scheduled CPU



Register renaming

- Reservation station and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - ⋆ Copied to reservation station
    - ⋆ No longer required in the register, can be overwritten
  - If operand is not yet available
    - ⋆ It will be provided to the reservation station by a function unit
    - ⋆ Register update may not be required

Why do dynamic scheduling

- Continue to execute instructions while waiting for stall to end
- Branch prediction
- Compilation and new hardware