MIPS ISA and Single Cycle Datapath

Computer Science 104

Outline of Today's Lecture

- Homework #5
- The MIPS Instruction Set
- Datapath and timing for Reg-Reg Operations
- Datapath for Logical Operations with Immediate
- Datapath for Load and Store Operations
- Datapath for Branch and Jump Operations

The MIPS Instruction Formats

° All MIPS instructions are 32 bits long. The three instruction formats:

• R-type

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

• I-type

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

• J-type

31 26 0

op	target address
6 bits	26 bits

° The different fields are:

- op: operation of the instruction
- rs, rt, rd: the source and destination register specifiers
- shamt: shift amount
- funct: selects the variant of the operation in the “op” field
- address / immediate: address offset or immediate value
- target address: target address of the jump instruction

The MIPS Subset (We can’t implement them all!)

° ADD and subtract

- add rd, rs, rt
- sub rd, rs, rt

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

° OR Immediate:

- ori rt, rs, imm16

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

° LOAD and STORE

- lw rt, rs, imm16
- sw rt, rs, imm16

° BRANCH:

- beq rs, rt, imm16

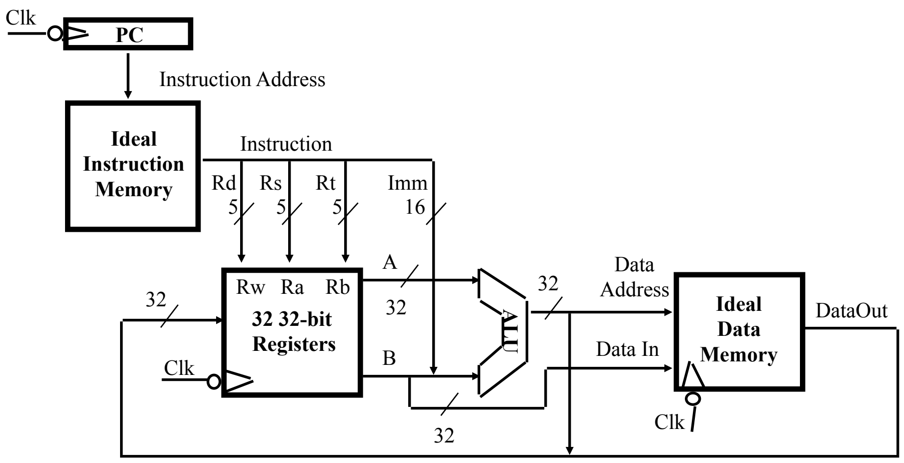
° JUMP:

- j target

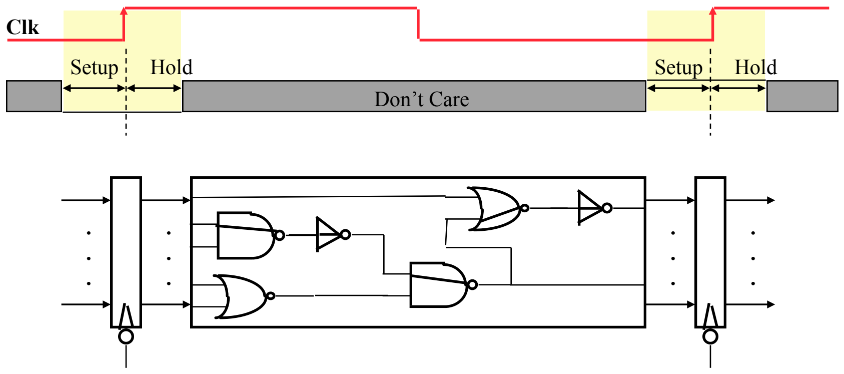
31 26 0

op	target address
6 bits	26 bits

An Abstract View of the Implementation



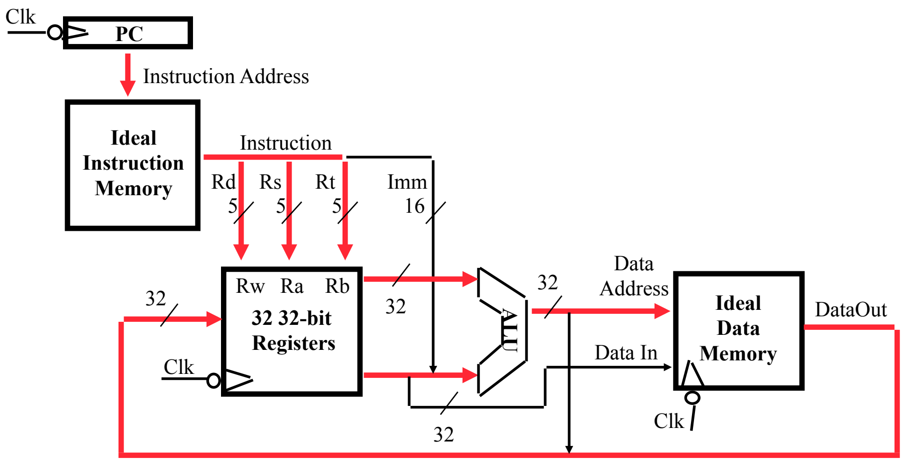
Clocking Methodology



- All storage elements are clocked by the same clock edge
- Cycle Time >= CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- Longest delay path = critical path

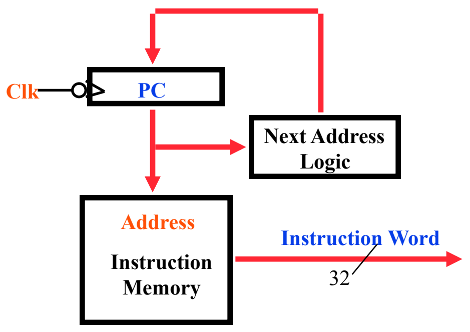
An Abstract View of the Critical Path

° Register file and ideal memory:
  • The CLK input is a factor ONLY during write operation
  • During read operation, behave as combinational logic:
    - Address valid => Output valid after "access time."



Overview of the Instruction Fetch Unit

° The common RTL operations
  • Fetch the Instruction: mem[PC]
  • Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- "something else"



RTL: The ADD Instruction

- add rd, rs, rt
  - mem[PC] Fetch the instruction from memory
  - R[rd] <- R[rs] + R[rt] The ADD operation
  - PC <- PC + 4 Calculate the next instruction’s address

RTL: The Load Instruction

- lw rt, rs, imm16
  - mem[PC] Fetch the instruction from memory
  - Address <- R[rs] + SignExt(imm16)
    Calculate the memory address
  - R[rt] <- Mem[Address] Load the data into the register
  - PC <- PC + 4 Calculate the next instruction’s address

RTL: The ADD Instruction

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

° add rd, rs, rt

• mem[PC]	Fetch the instruction from memory
• R[rd] <- R[rs] + R[rt]	The actual operation
• PC <- PC + 4	Calculate the next instruction's address

RTL: The Subtract Instruction

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

° sub rd, rs, rt

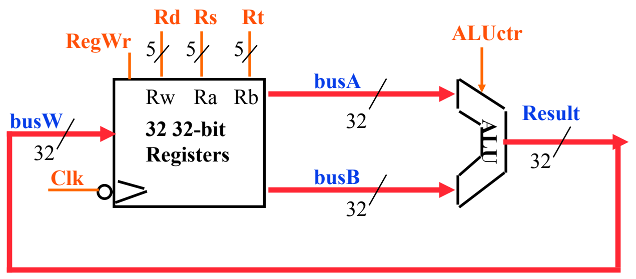
• mem[PC]	Fetch the instruction from memory
• R[rd] <- R[rs] - R[rt]	The actual operation
• PC <- PC + 4	Calculate the next instruction's address

Datapath for Register-Register Operations

° R[rd] <- R[rs] op R[rt]   Example: add   rd, rs, rt

- Ra, Rb, and Rw comes from instruction's rs, rt, and rd fields
- ALUctr and RegWr: control logic after decoding the instruction fields: op and func

31 … 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



RTL: The OR Immediate Instruction

31 … 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

° ori   rt, rs, imm16

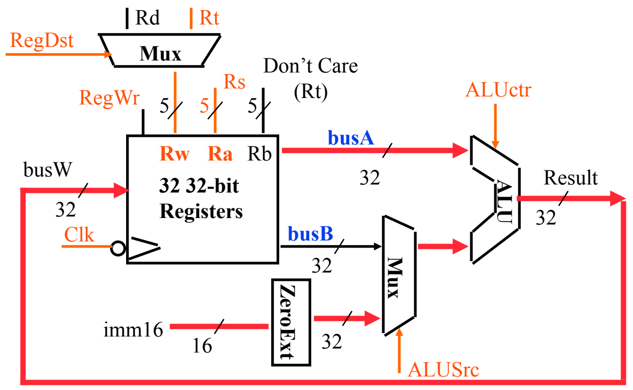
- mem[PC]   Fetch the instruction from memory
- R[rt] <- R[rs] or ZeroExt(imm16)   The OR operation
- PC <- PC + 4   Calculate the next instruction's address

31 … 16	15 … 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	immediate
16 bits	16 bits

Datapath for Logical Operations with Immediate

° R[rt] <- R[rs] op ZeroExt[imm16]]   Example: ori   rt, rs, imm16

31	26	21	16	0
op	rs	rt	immediate	
6 bits	5 bits	5 bits	16 bits	



RTL: The Load Instruction

31	26	21	16	0
op	rs	rt	immediate	
6 bits	5 bits	5 bits	16 bits	

° lw   rt, rs, imm16

- mem[PC]   Fetch the instruction from memory
- Address <- R[rs] + SignExt(imm16)

  Calculate the memory address

  R[rt] <- Mem[Address]   Load the data into the register
- PC <- PC + 4   Calculate the next instruction's address

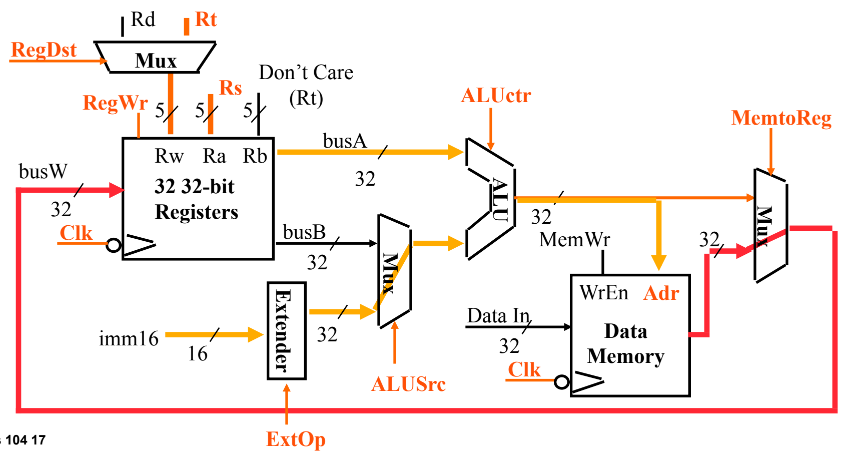
31	16	15	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		0 immediate	
16 bits		16 bits	

31	16	15	0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1		1 immediate	
16 bits		16 bits	

Datapath for Load Operations

° R[rt] <- Mem[R[rs] + SignExt[imm16]]   Example: lw   rt, rs, imm16

31 – 26	25 – 21	20 – 16	15 – 0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits



RTL: The Store Instruction

31 – 26	25 – 21	20 – 16	15 – 0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

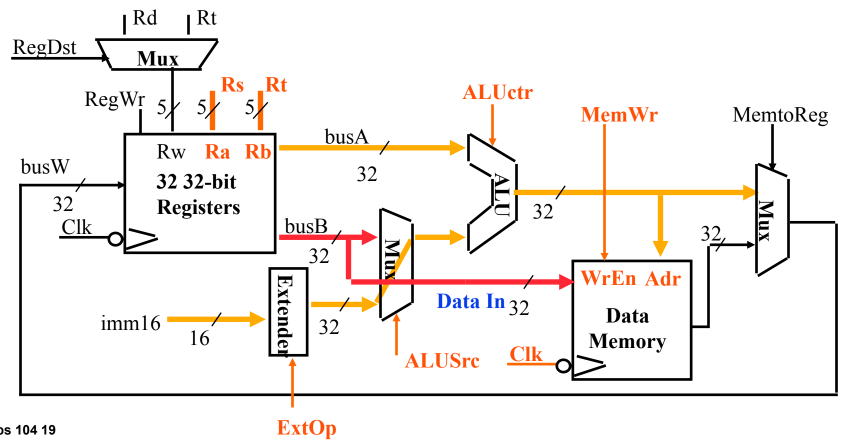
° sw   rt, rs, imm16

- mem[PC]   **Fetch the instruction from memory**
- Address <- R[rs] + SignExt(imm16)   **Calculate the memory address**
- Mem[Address] <- R[rt]   **Store the register into memory**
- PC <- PC + 4   **Calculate the next instruction's address**

Datapath for Store Operations

° Mem[R[rs] + SignExt[imm16] <- R[rt]]   Example: sw   rt, rs, imm16

31	26 21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits



RTL: The Branch Instruction

31	26 21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

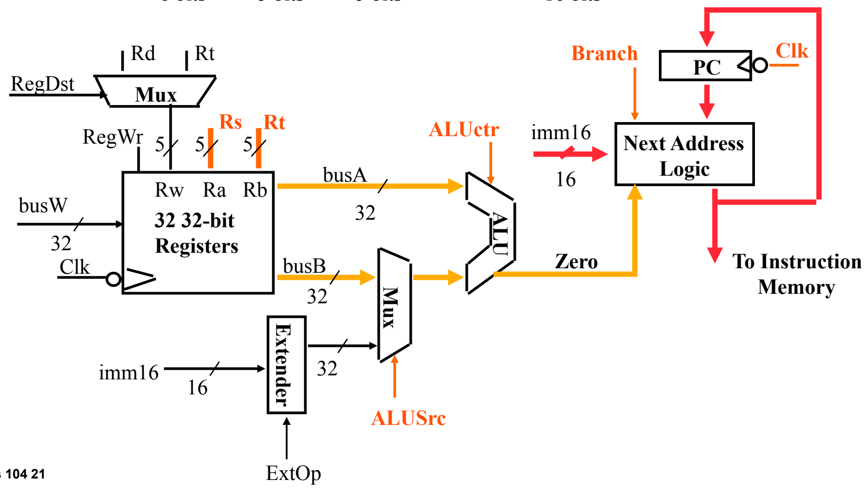
° beq   rs, rt, imm16

- mem[PC]   Fetch the instruction from memory
- Cond <- R[rs] - R[rt]   Calculate the branch condition
- if (COND eq 0)   Calculate the next instruction's address
  PC relative branches (no condition codes)
  - PC <- PC + 4 + (SignExt(imm16) x 4)
- else
  - PC <- PC + 4

Datapath for Branch Operations

° **beq rs, rt, imm16** We need to compare Rs and Rt!

31 — 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits



Binary Arithmetic for the Next Address

° In theory, the PC is a 32-bit byte address into the instruction memory:
- Sequential operation: PC<31:0> = PC<31:0> + 4
- Branch operation: PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] * 4

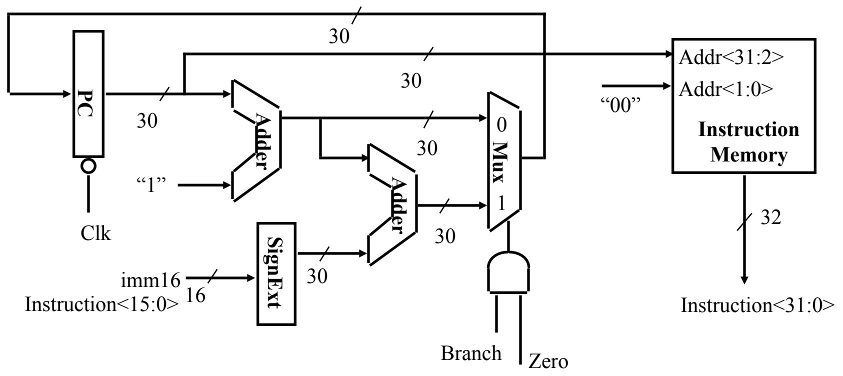
° The magic number "4" always comes up because:
- The 32-bit PC is a byte address
- And all our instructions are 4 bytes (32 bits) long

° In other words:
- The 2 LSBs of the 32-bit PC are always zeros
- There is no reason to have hardware to keep the 2 LSBs

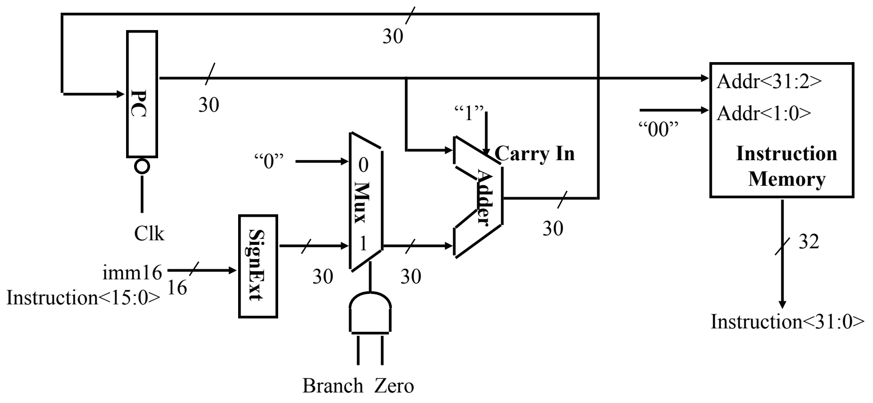
° In practice, we can simplify the hardware by using a 30-bit PC<31:2>:
- Sequential operation: PC<31:2> = PC<31:2> + 1
- Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
- In either case: Instruction-Memory-Address = PC<31:2> concat "00"

Next Address Logic: Expensive and Fast Solution

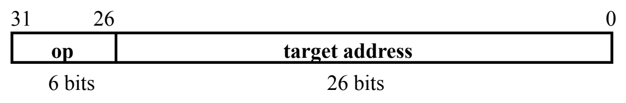
- Using a 30-bit PC:
  - Sequential operation: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - In either case: Instruction-Memory-Address = PC<31:2> concat "00"



Next Address Logic



RTL: The Jump Instruction



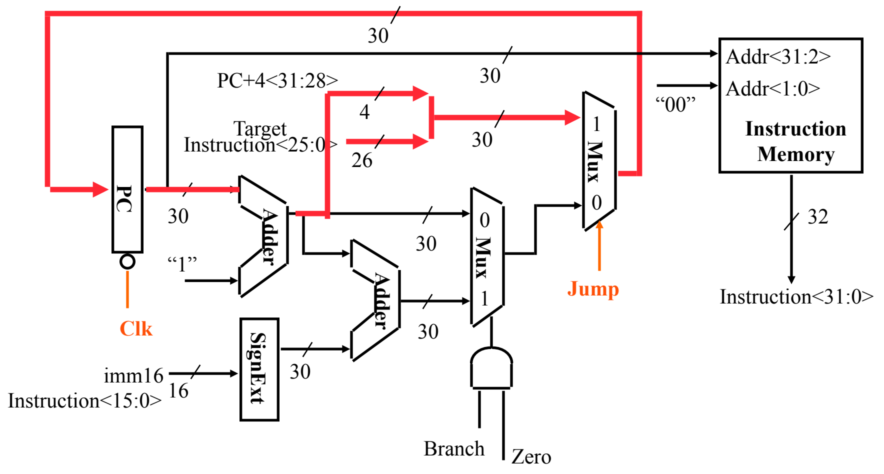
° j target

- mem[PC] Fetch the instruction from memory
- PC <- PC+4<31:28> concat target<25:0> concat <00>
  Calculate the next instruction's address

Instruction Fetch Unit

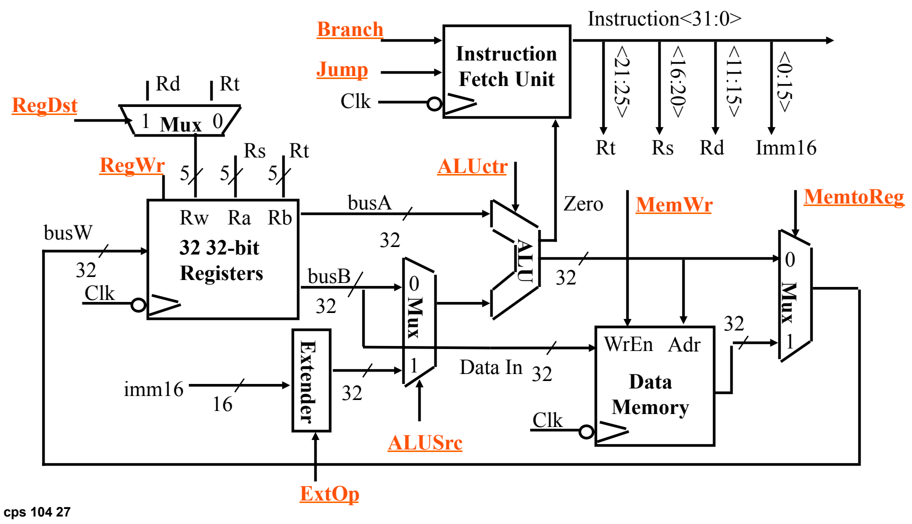
° j target

- PC<31:2> <- PC+4<31:28> concat target<25:0>



Putting it All Together: A Single Cycle Datapath

° We have everything except control signals.



Recap: The MIPS Instruction Formats

° All MIPS instructions are 32 bits long. The three instruction formats:

• R-type

31 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

• I-type

31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

• J-type

31 26	0
op	target address
6 bits	26 bits

° The different fields are:

- op: operation of the instruction
- rs, rt, rd: the source and destination registers specifier
- shamt: shift amount
- funct: selects the variant of the operation in the “op” field
- address / immediate: address offset or immediate value
- target address: target address of the jump instruction

Recap: The MIPS Subset

° ADD and subtract
  • add rd, rs, rt
  • sub rd, rs, rt

31 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

° OR Imm:
  • ori rt, rs, imm16

31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

° LOAD and STORE
  • lw rt, rs, imm16
  • sw rt, rs, imm16

° BRANCH:
  • beq rs, rt, imm16

° JUMP:
  • j target

31 26	0
op	target address
6 bits	26 bits

RTL: The ADD Instruction

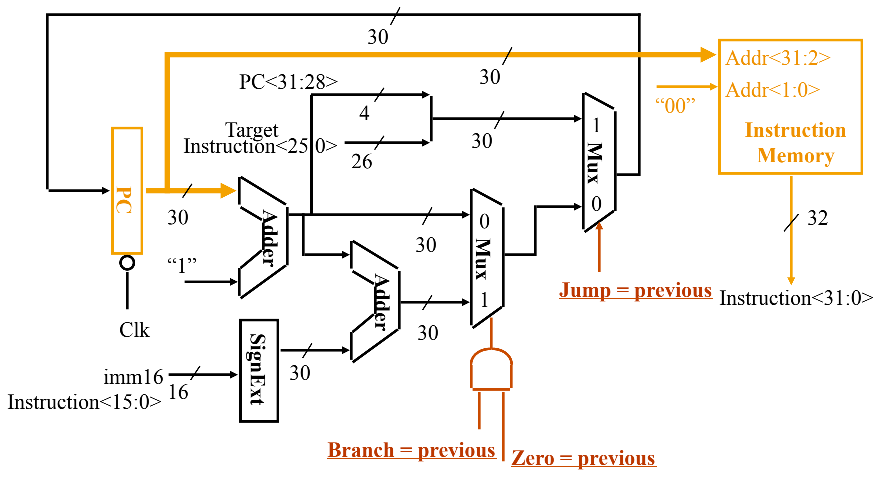
31 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

° add   rd, rs, rt

• mem[PC]	Fetch the instruction from memory
• R[rd] <- R[rs] + R[rt]	The actual operation
• PC <- PC + 4	Calculate the next instruction's address

Instruction Fetch Unit at the Beginning of Add / Subtract

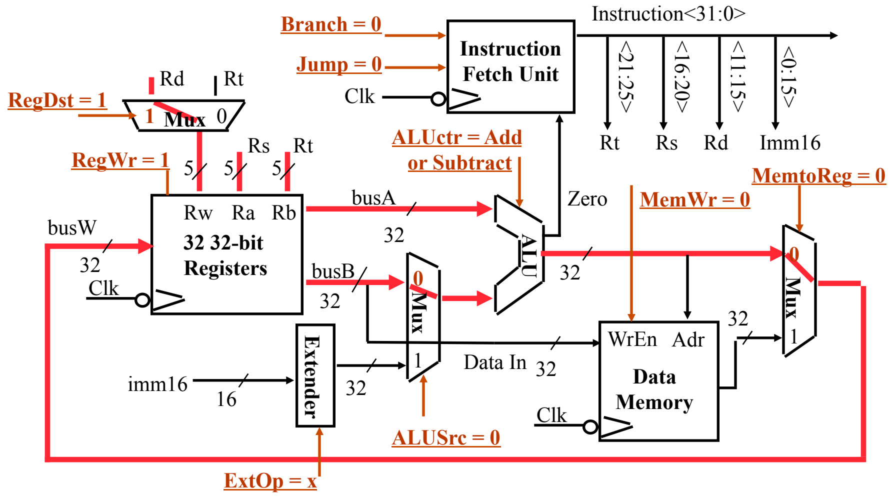
° Fetch the instruction from Instruction memory: Instruction <- mem[PC]
  • This is the same for all instructions



The Single Cycle Datapath during Add and Subtract

31 26	26 21	21 16	16 11	11 6	6 0
op	rs	rt	rd	shamt	funct

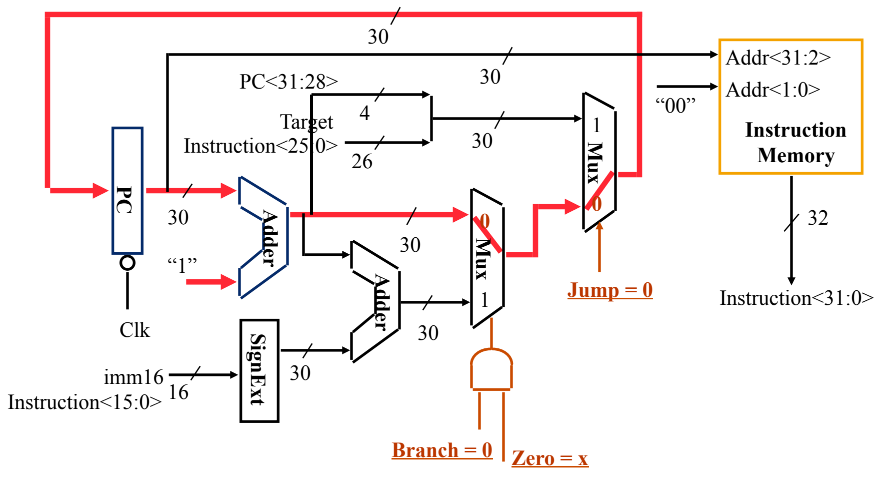
° R[rd] <- R[rs] + / - R[rt]



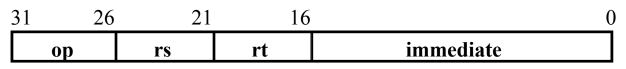
Instruction Fetch Unit at the End of Add and Subtract

° PC <- PC + 4

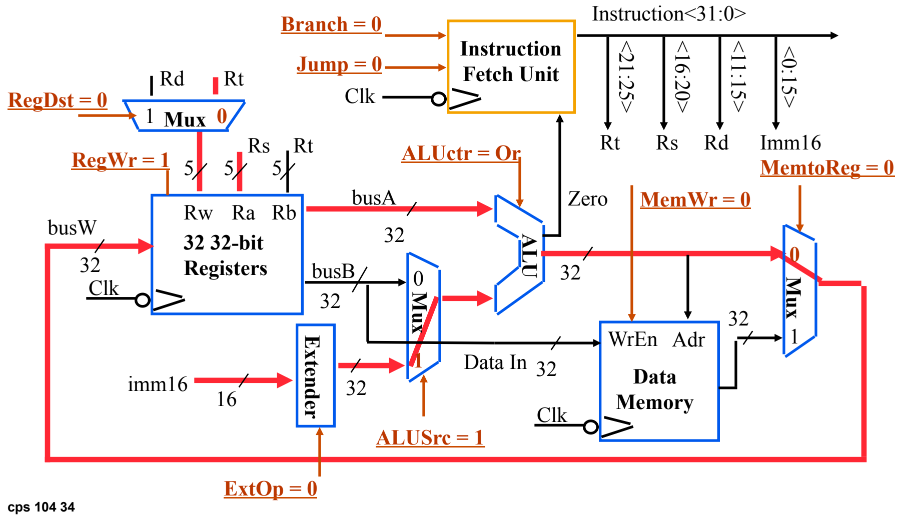
- This is the same for all instructions except: Branch and Jump



The Single Cycle Datapath during Or Immediate



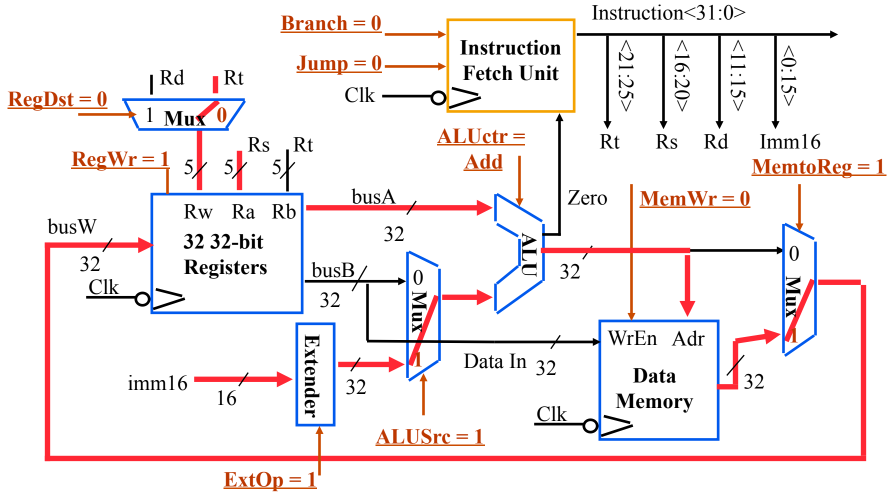
° R[rt] <- R[rs] or ZeroExt[Imm16]



The Single Cycle Datapath during Load

31	26	21	16	0
op	rs	rt	immediate	

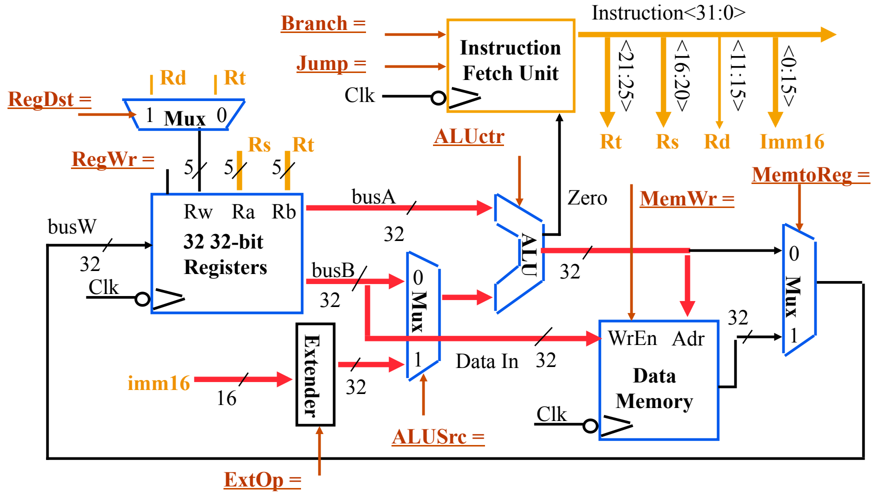
- R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



The Single Cycle Datapath during Store (fill it in)

31	26	21	16	0
op	rs	rt	immediate	

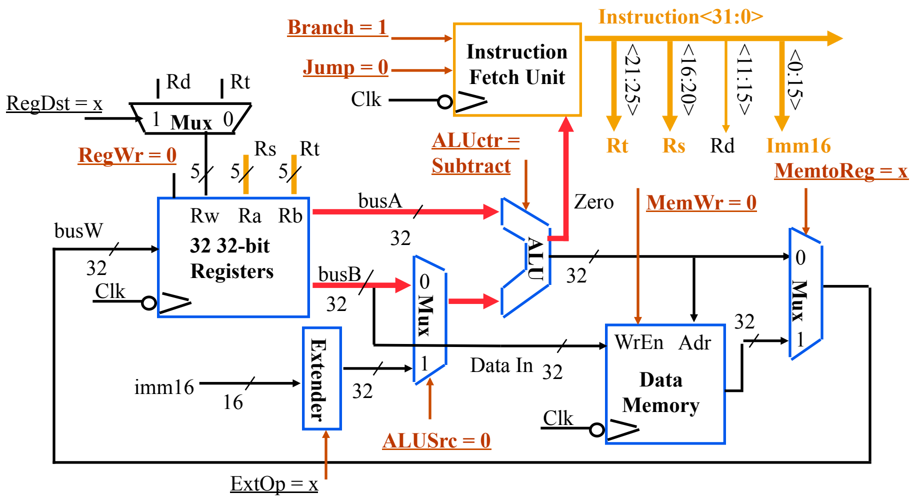
- Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



The Single Cycle Datapath during Branch

31	26	21	16	0
op	rs	rt	immediate	

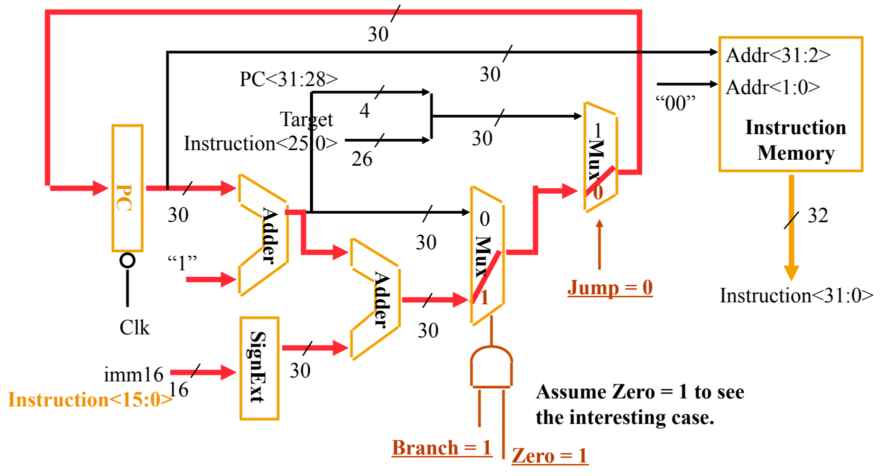
° if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



Instruction Fetch Unit at the End of Branch

31	26	21	16	0
op	rs	rt	immediate	

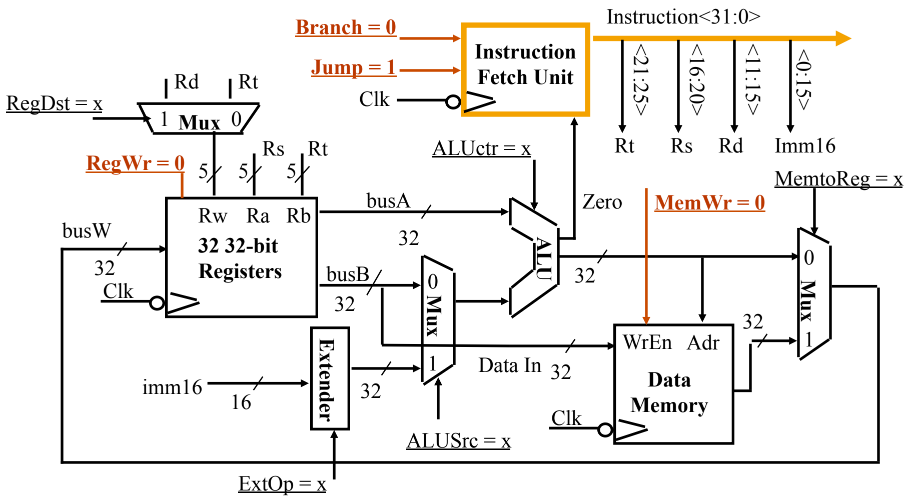
° if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4



The Single Cycle Datapath during Jump

31	26	0
op	target address	

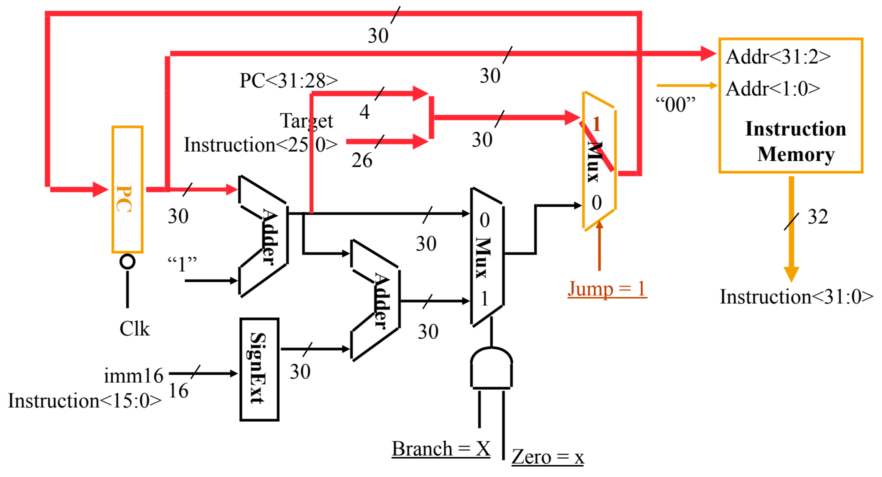
° Nothing to do! Make sure control signals are set correctly!



Instruction Fetch Unit at the End of Jump

31	26	0
op	target address	

° PC <- PC<31:28> concat target<25:0> concat “00”



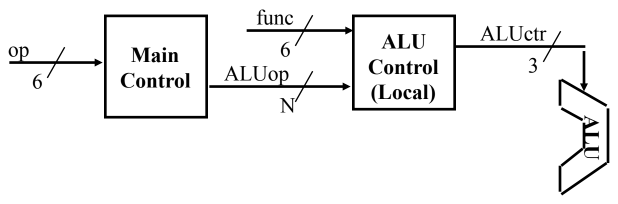
A Summary of the Control Signals

func	10 0000	10 0010	We Don't Care :-)				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	**sub**	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	x
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

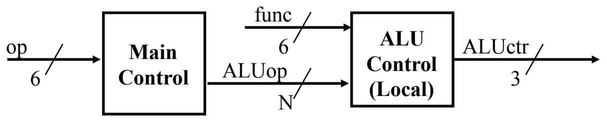
	31	26	21	16	11	6	0	
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**		**add, sub**
I-type	**op**	**rs**	**rt**	**immediate**				**ori, lw, sw, beq**
J-type	**op**	**target address**						**jump**

The Concept of Local Decoding

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	x
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



The Encoding of ALUop

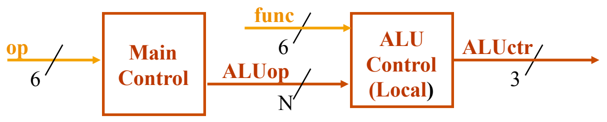


° In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract

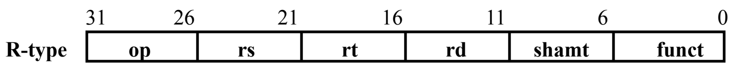
° To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

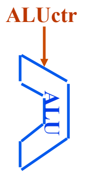
Decoding the "func" Field



	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx



func<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than



ALUctr<2:0>	ALU Operation
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

The Truth Table for ALUctr

funct<3:0>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop (Symbolic)	R-type	ori	lw	sw	beq
	"R-type"	Or	Add	Add	Subtract
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01

ALUop			func				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

The Logic Equation for ALUctr<2>

ALUop			func				ALUctr<2>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

This makes func<3> a don't care

° ALUctr<2> = !ALUop<2> & ALUop<0> +
ALUop<2> & !func<2> & func<1> & !func<0>

The Logic Equation for ALUctr<1>

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<1>
0	0	0	x	x	x	x	1
0	x	1	x	x	x	x	1
1	x	x	0	0	0	0	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

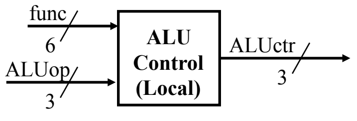
° ALUctr<1> = !ALUop<2> & !ALUop<1> +
ALUop<2> & !func<2> & !func<0>

The Logic Equation for ALUctr<0>

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
0	1	x	x	x	x	x	1
1	x	x	0	1	0	1	1
1	x	x	1	0	1	0	1

° ALUctr<0> = !ALUop<2> & ALUop<0>
+ ALUop<2> & !func<3> & func<2> & !func<1> & func<0>
+ ALUop<2> & func<3> & !func<2> & func<1> & !func<0>

The ALU Control Block

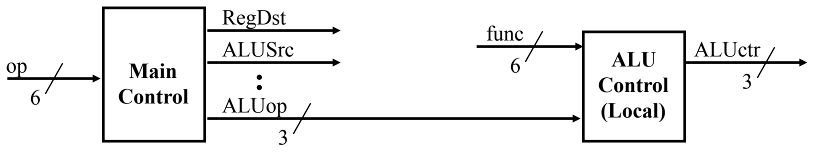


° ALUctr<2> = !ALUop<2> & ALUop<0> +
ALUop<2> & !func<2> & func<1> & !func<0>

° ALUctr<1> = !ALUop<2> & !ALUop<1> +
ALUop<2> & !func<2> & !func<0>

° ALUctr<0> = !ALUop<2> & ALUop<0>
+ ALUop<2> & !func<3> & func<2> & !func<1> & func<0>
+ ALUop<2> & func<3> & !func<2> & func<1> & !func<0>

The "Truth Table" for the Main Control (rotated)



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	x
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

The “Truth Table” for RegWrite

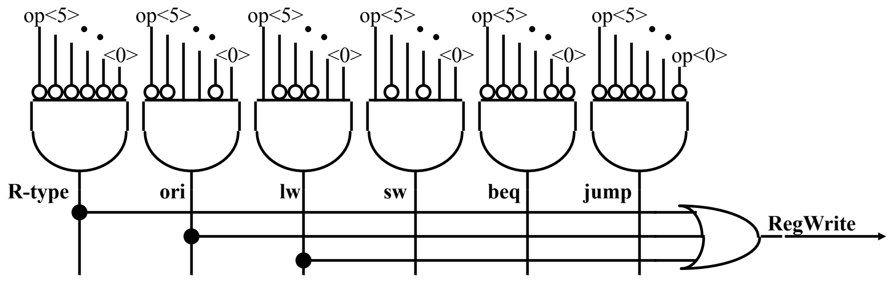
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

° RegWrite = R-type + ori + lw

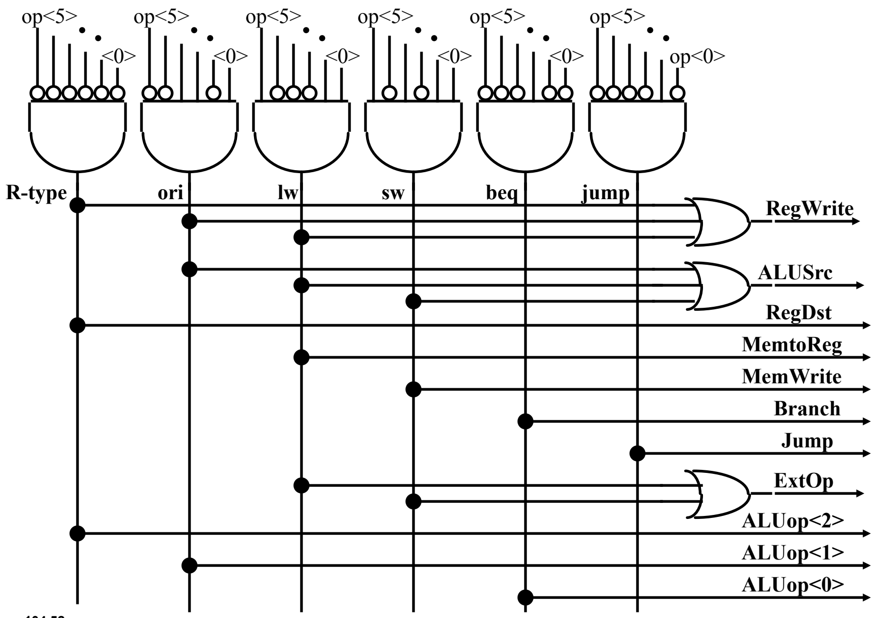
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)

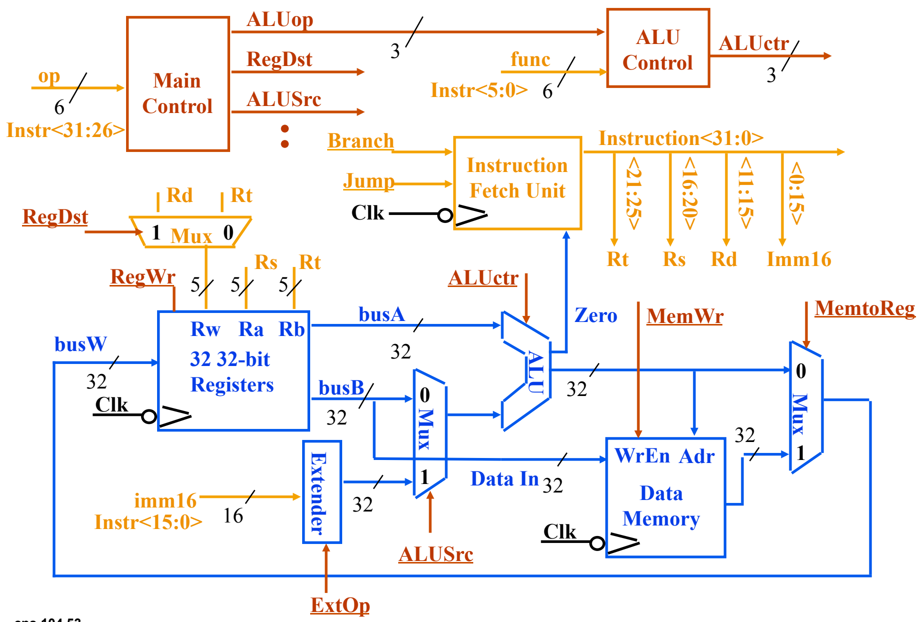
+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



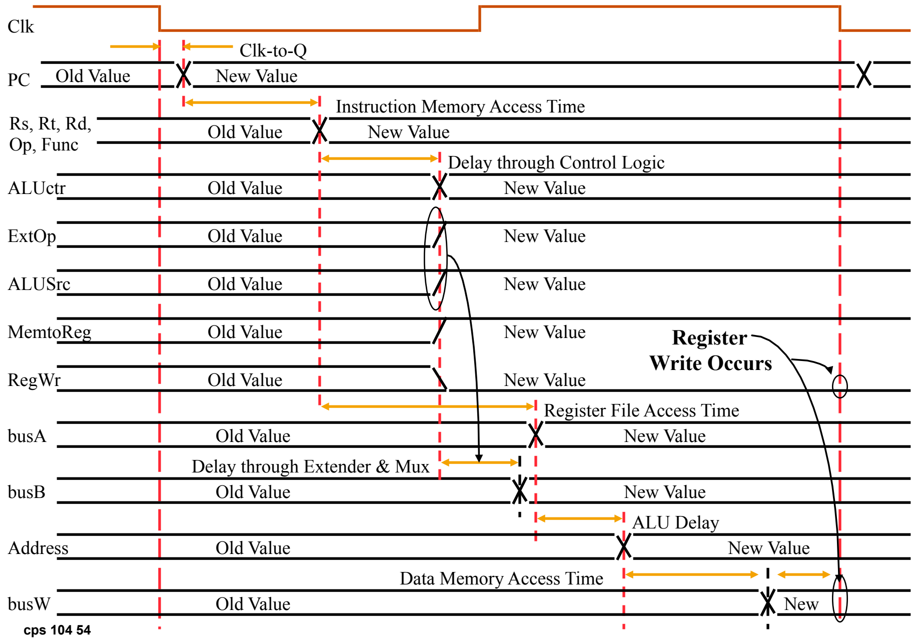
Implementation of the Main Control



Putting it All Together: A Single Cycle Processor



Worst Case Timing: lw $1, $2(offset)



Drawback of this Single Cycle Processor

- Long cycle time:
  - Cycle time must be long enough for the load instruction:

    PC's Clock -to-Q +
    Instruction Memory Access Time +
    Register File Access Time +
    ALU Delay (address calculation) +
    Data Memory Access Time +
    Register File Setup Time +
    Clock Skew
- Cycle time is much longer than needed for all other instructions

Summary

- What's ahead
  - Pipelined processors
  - Memory
  - Input / Output