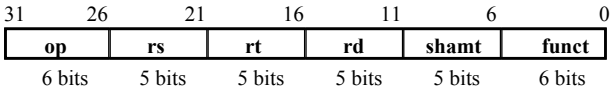
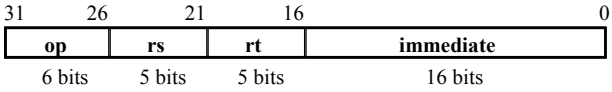
361
Computer Architecture
Lecture 9: Designing Single Cycle Control

Recap: The MIPS Subset

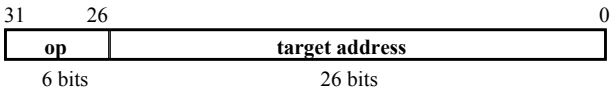
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt



- OR Imm:
  - ori rt, rs, imm16

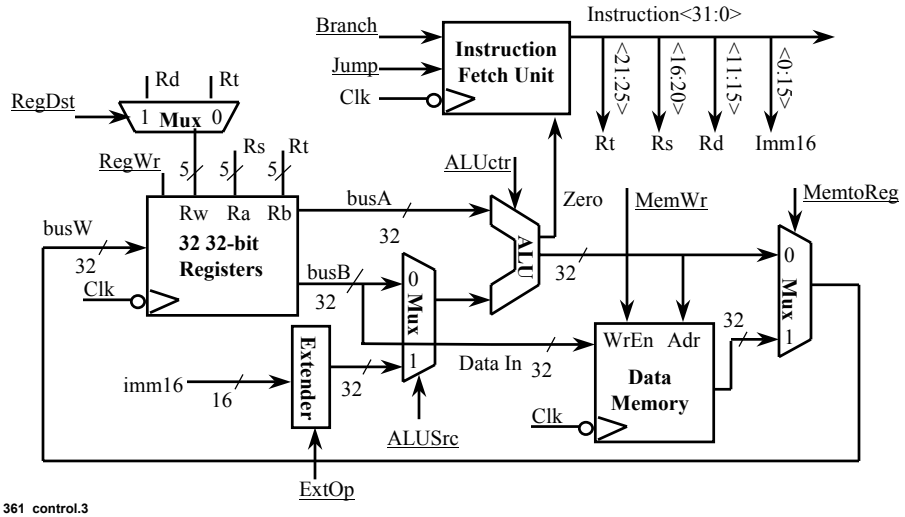


- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



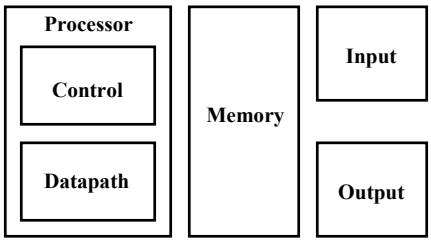
Recap: A Single Cycle Datapath

° We have everything except control signals (underline)
  • Today's lecture will show you how to generate the control signals



The Big Picture: Where are We Now?

° The Five Classic Components of a Computer

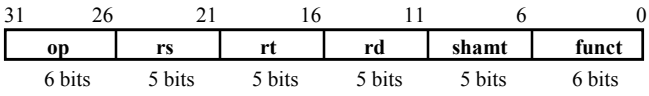


° Today's Topic: Designing the Control for the Single Cycle Datapath

Outline of Today's Lecture

- Recap and Introduction
- Control for Register-Register & Or Immediate instructions
- Control signals for Load, Store, Branch, & Jump
- Building a local controller: ALU Control
- The main controller
- Summary

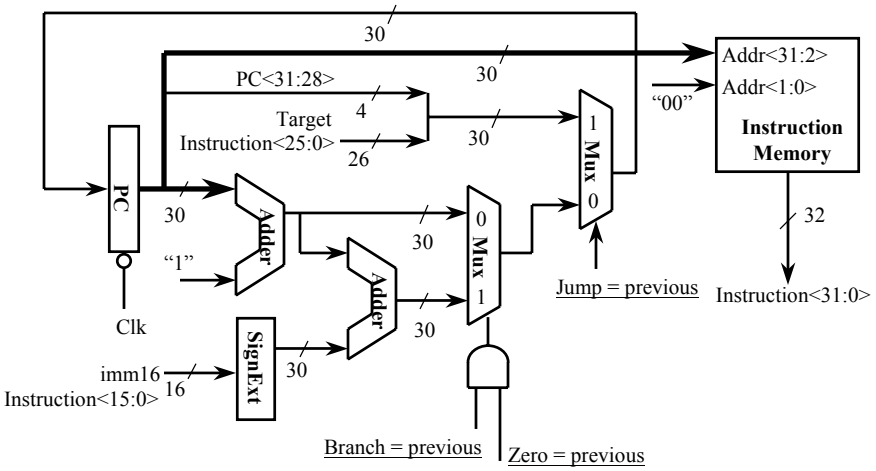
RTL: The ADD Instruction



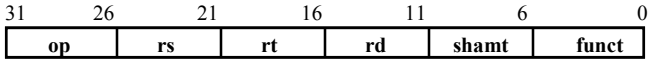
- add rd, rs, rt
  - mem[PC] — Fetch the instruction from memory
  - R[rd] <- R[rs] + R[rt] — The actual operation
  - PC <- PC + 4 — Calculate the next instruction's address

Instruction Fetch Unit at the Beginning of Add / Subtract

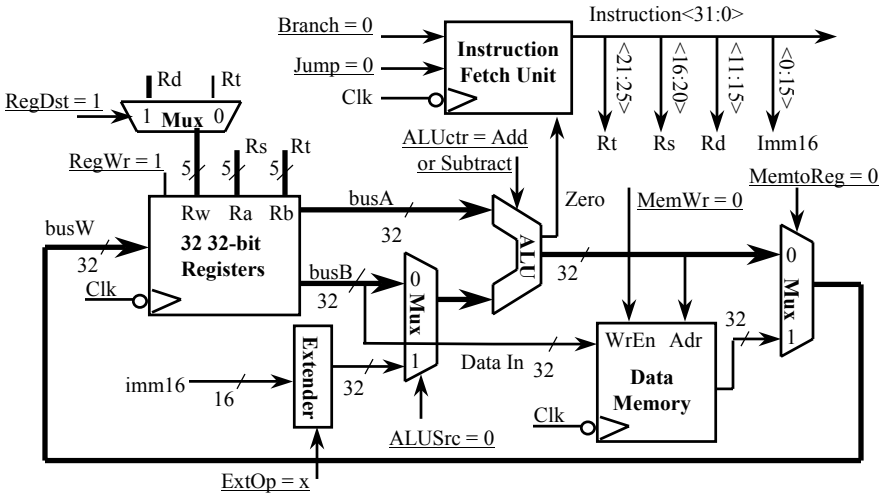
° Fetch the instruction from Instruction memory: Instruction <- mem[PC]
  • This is the same for all instructions



The Single Cycle Datapath during Add and Subtract



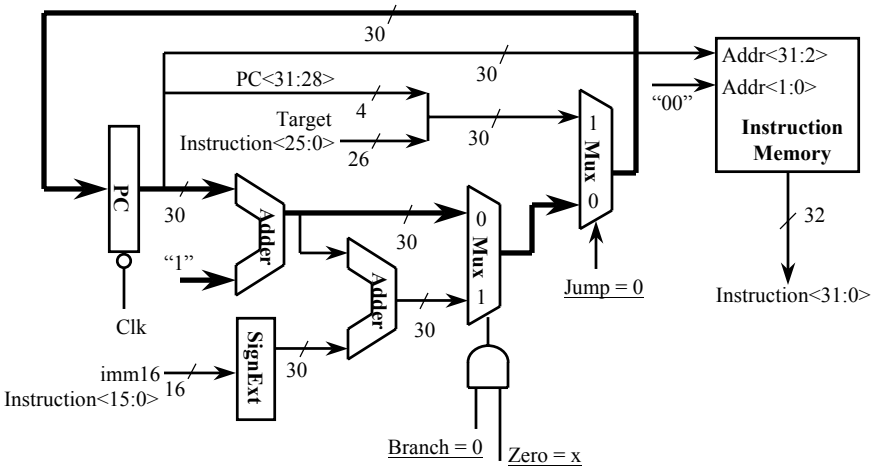
° R[rd] <- R[rs] + / - R[rt]



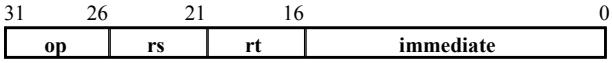
Instruction Fetch Unit at the End of Add and Subtract

° **PC <- PC + 4**

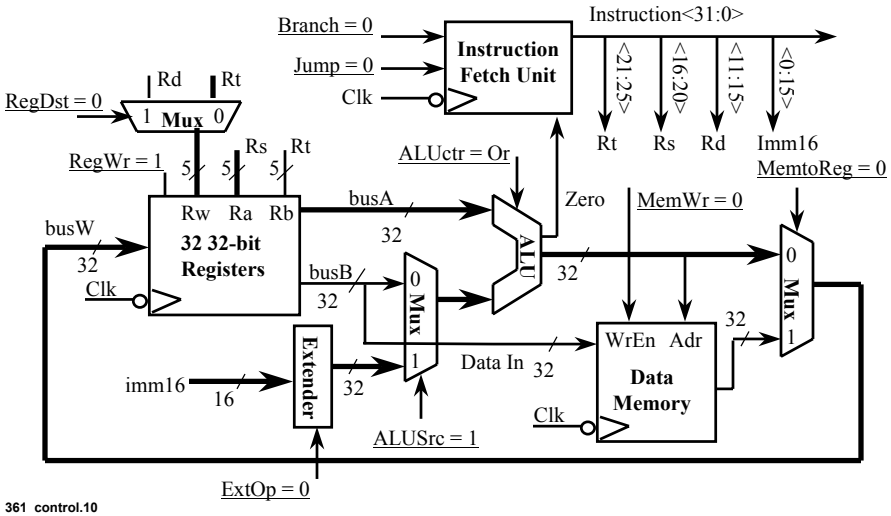
- **This is the same for all instructions except: Branch and Jump**



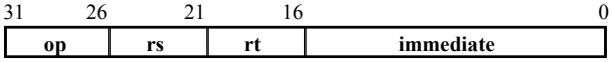
The Single Cycle Datapath during Or Immediate



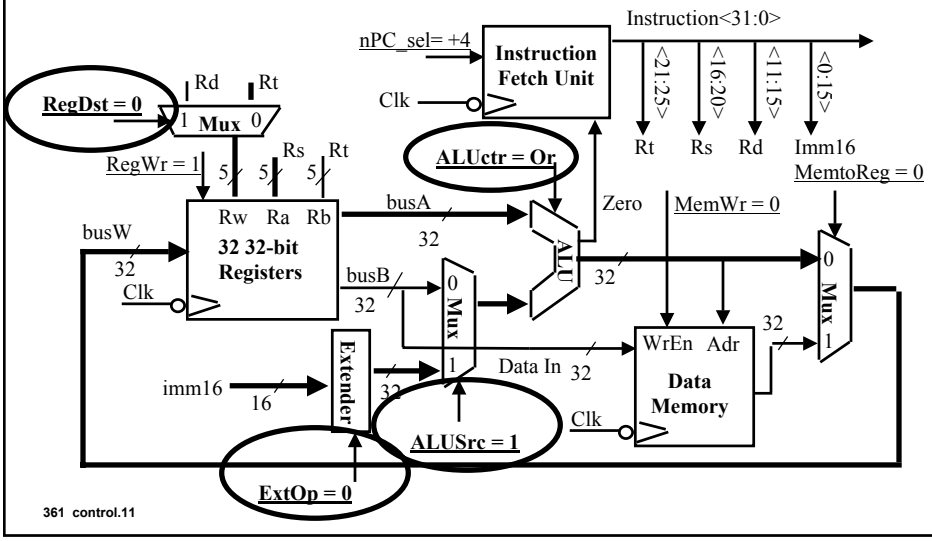
° **R[rt] <- R[rs] or ZeroExt[Imm16]**



The Single Cycle Datapath during Or Immediate

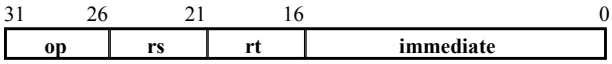


° R[rt] <- R[rs] or ZeroExt[Imm16]

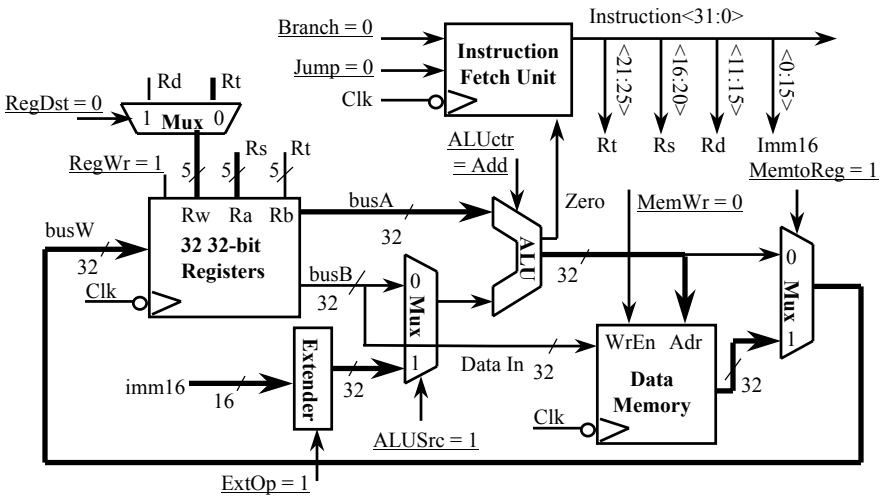


361 control.11

The Single Cycle Datapath during Load

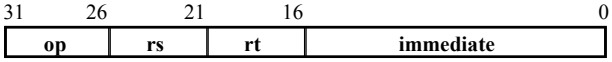


° R[rt] <- Data Memory {R[rs] + SignExt[imm16]}

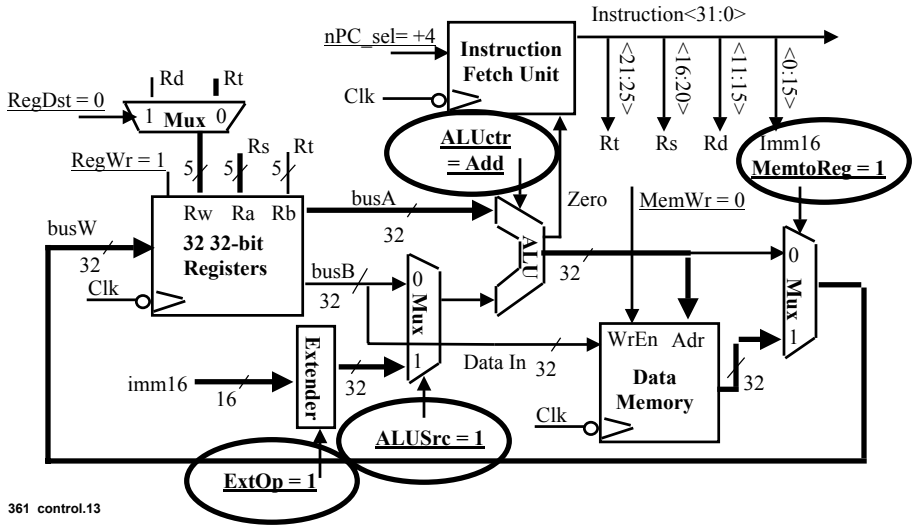


361 control.12

The Single Cycle Datapath during Load

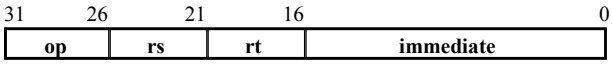


° R[rt] <- Data Memory {R[rs] + SignExt[imm16]}

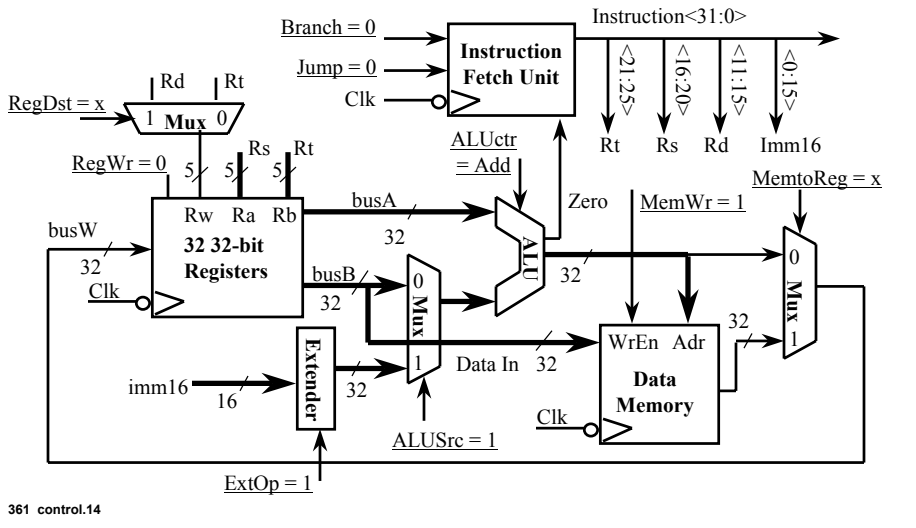


361 control.13

The Single Cycle Datapath during Store

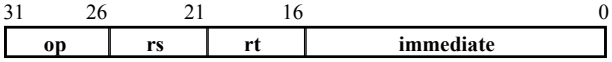


° Data Memory {R[rs] + SignExt[imm16]} <- R[rt]

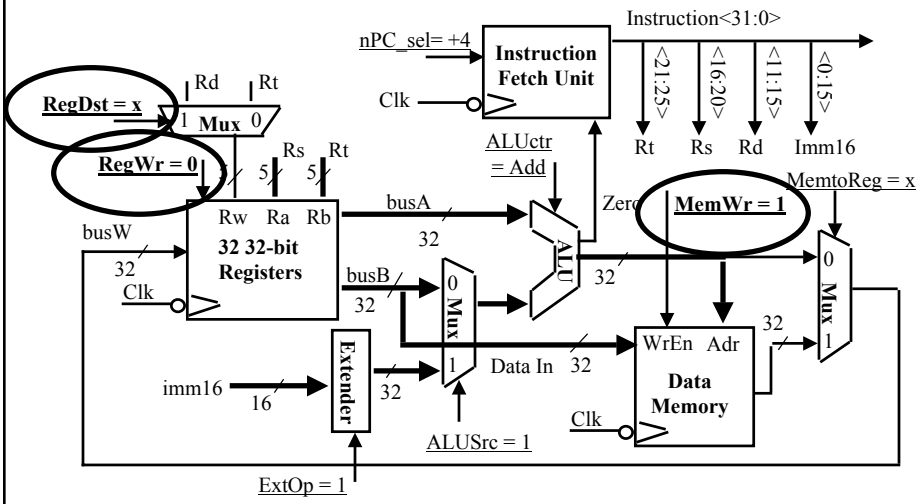


361 control.14

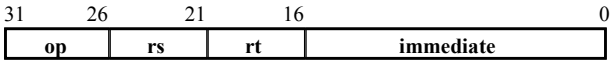
The Single Cycle Datapath during Store



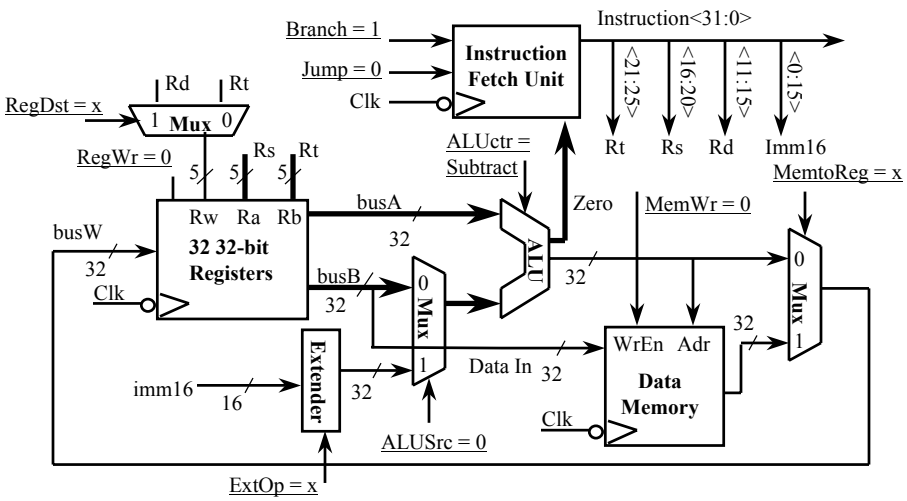
° Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



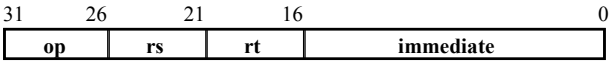
The Single Cycle Datapath during Branch



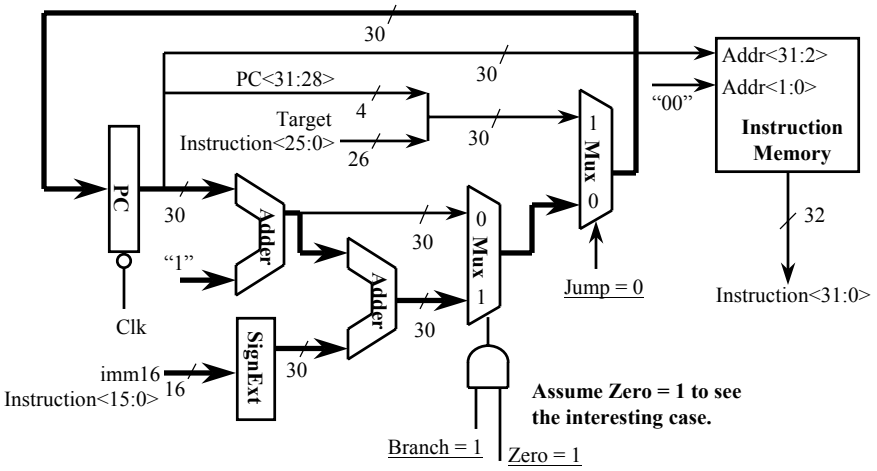
° if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



Instruction Fetch Unit at the End of Branch

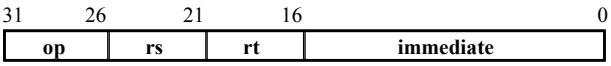


° if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4

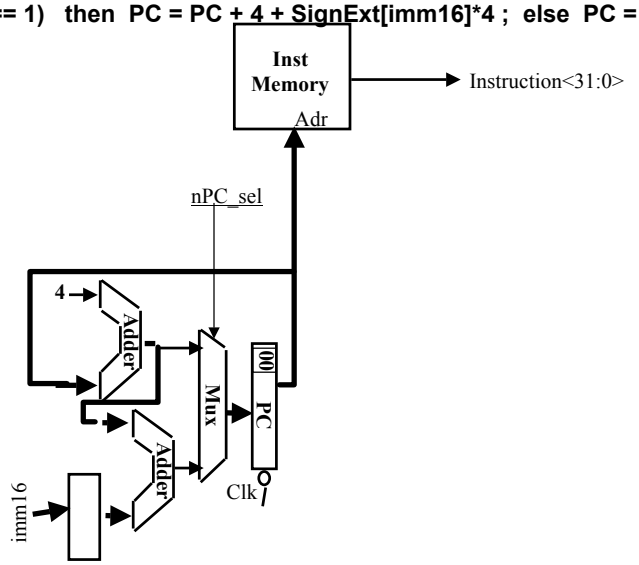


361 control.17

Instruction Fetch Unit at the End of Branch

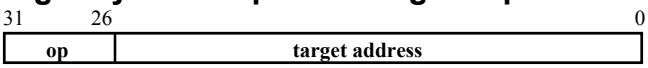


° if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4

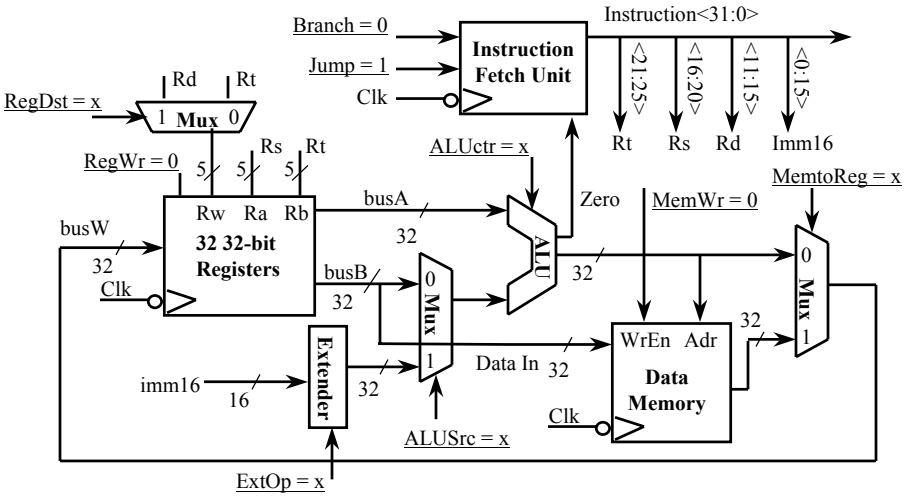


361 control.18

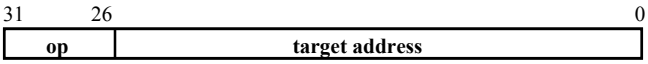
The Single Cycle Datapath during Jump



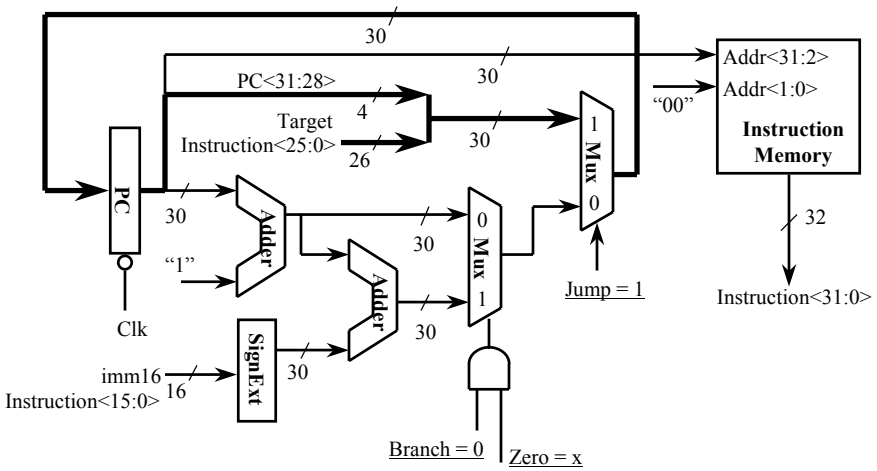
° Nothing to do! Make sure control signals are set correctly!



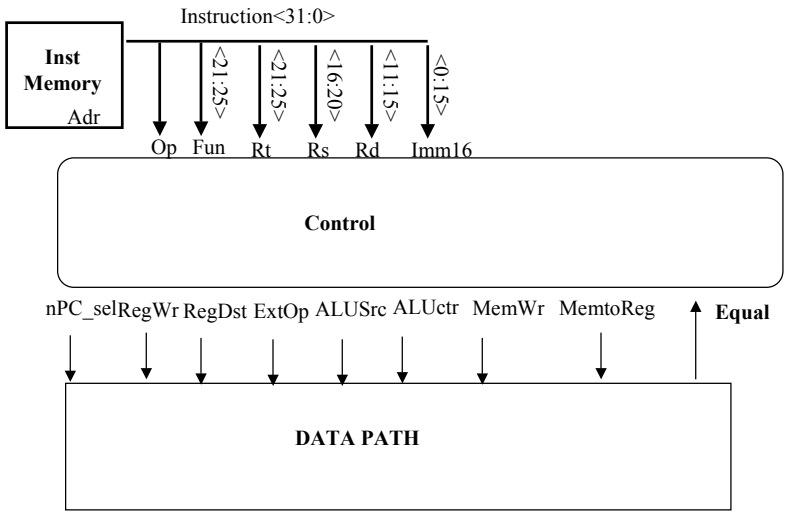
Instruction Fetch Unit at the End of Jump



° PC <- PC<31:29> concat target<25:0> concat “00”



Step 4: Given Datapath: RTL -> Control

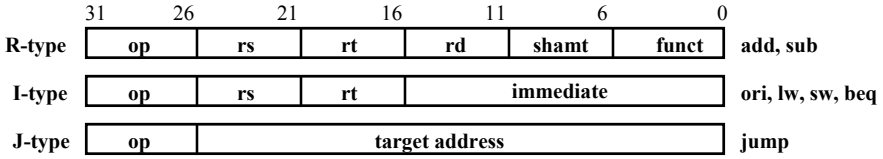


A Summary of Control Signals

inst	Register Transfer	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”	
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”	
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
	ALUsrc = Im, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel = “+4”	
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr,	nPC_sel = “+4”
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, nPC_sel = “+4”	
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4	
	nPC_sel = “Br”, ALUctr = “sub”	

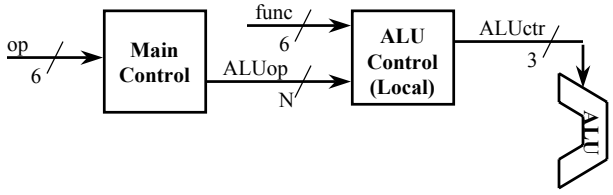
A Summary of the Control Signals

See Appendix A							
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	0	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	0
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

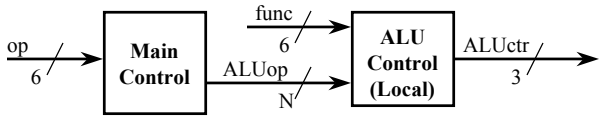


The Concept of Local Decoding

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	0
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

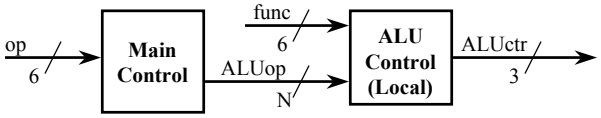


° In this exercise, ALUop has to be 2 bits wide to represent:
- (1) "R-type" instructions
- "I-type" instructions that require the ALU to perform:
  - (2) Or, (3) Add, and (4) Subtract

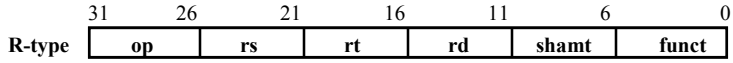
° To implement the full MIPS ISA, ALUop hat to be 3 bits to represent:
- (1) "R-type" instructions
- "I-type" instructions that require the ALU to perform:
  - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

	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

The Decoding of 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



(P. 286 text)

funct<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	Add
001	Subtract
010	And
110	Or
111	Set-on-less-than

The Truth Table for ALUctr

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

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

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

X Y Z A B C D

This makes func<3> a don’t care

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

= !X&Y + X&!A&!B&C&!D + X&A&!B&C&!D
= !X&Y + X&!B&C&!D

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

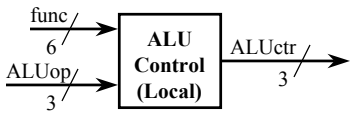
° ALUctr<1> = !ALUop<2> & !ALUop<0> +
ALUop<2> & !func<2> & !func<0>

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<0> +
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>

Step 5: Logic for each control signal

° nPC_sel <= if (OP == BEQ) then EQUAL else 0

° ALUsrc <= if (OP == “Rtype”) then “regB” else “immed”

° ALUctr <= if (OP == “Rtype”) then funct
elseif (OP == ORi) then “OR”
elseif (OP == BEQ) then “sub”
else “add”

° ExtOp <= _____

° MemWr <= _____

° MemtoReg <= _____

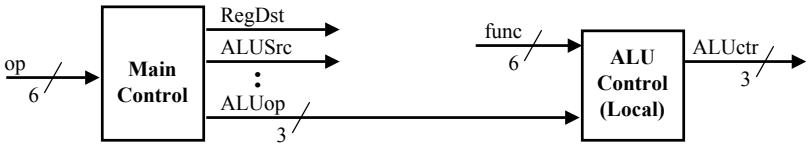
° RegWr: <=_____

° RegDst: <= _____

Step 5: Logic for each control signal

- **nPC_sel** <= if (OP == BEQ) then EQUAL else 0
- **ALUsrc** <= if (OP == “Rtype”) then “regB” else “immed”
- **ALUctr** <= if (OP == “Rtype”) then funct
  elseif (OP == ORi) then “OR”
  elseif (OP == BEQ) then “sub”
  else “add”
- **ExtOp** <= if (OP == ORi) then “zero” else “sign”
- **MemWr** <= (OP == Store)
- **MemtoReg** <= (OP == Load)
- **RegWr:** <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- **RegDst:** <= if ((OP == Load) || (OP == ORi)) then 0 else 1

The “Truth Table” for the Main Control



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	0
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

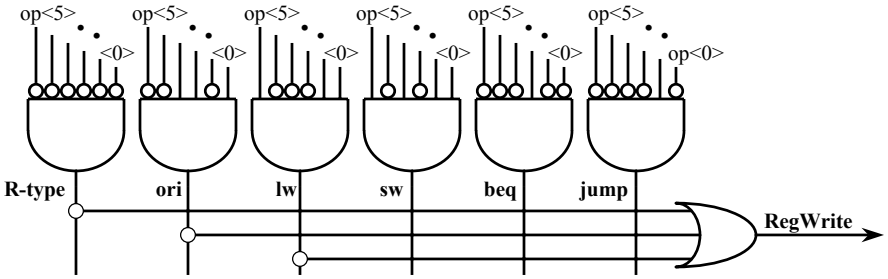
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	x	x	x

° **RegWrite = R-type + ori + lw**

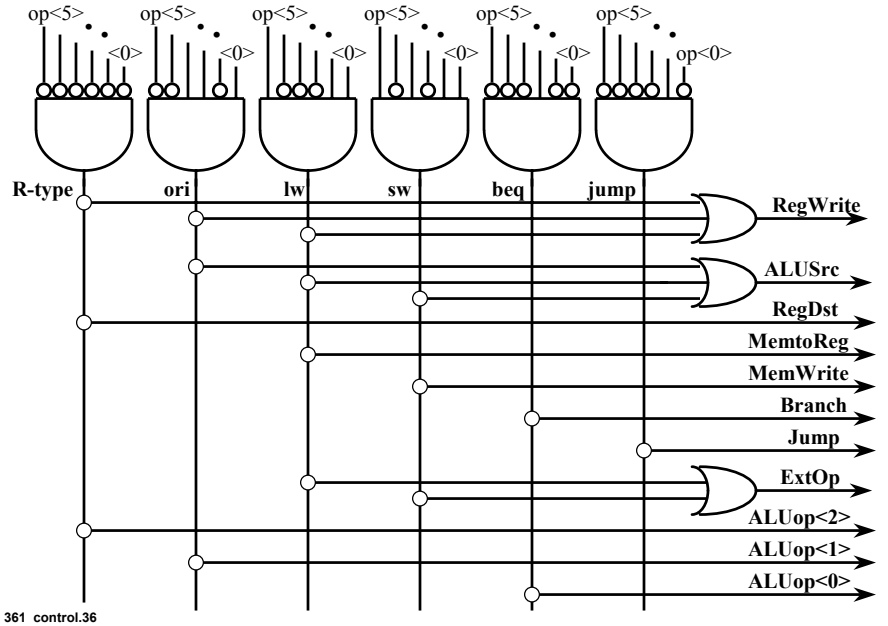
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> **(R-type)**

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

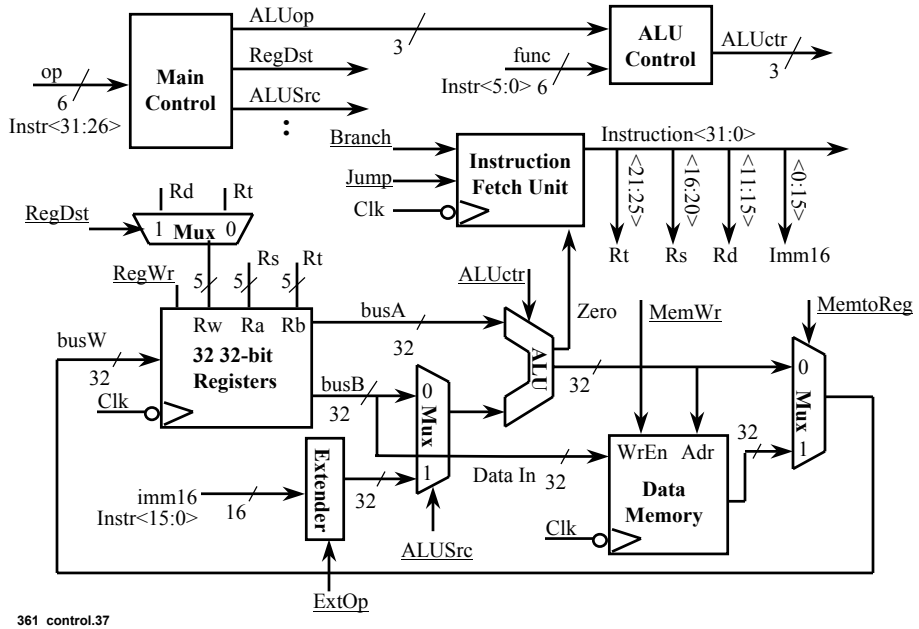
+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> **(lw)**



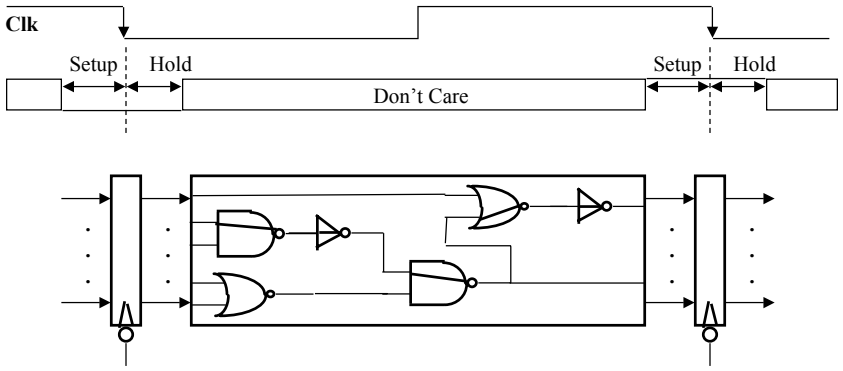
PLA Implementation of the Main Control



Putting it All Together: A Single Cycle Processor



Clocking Methodology

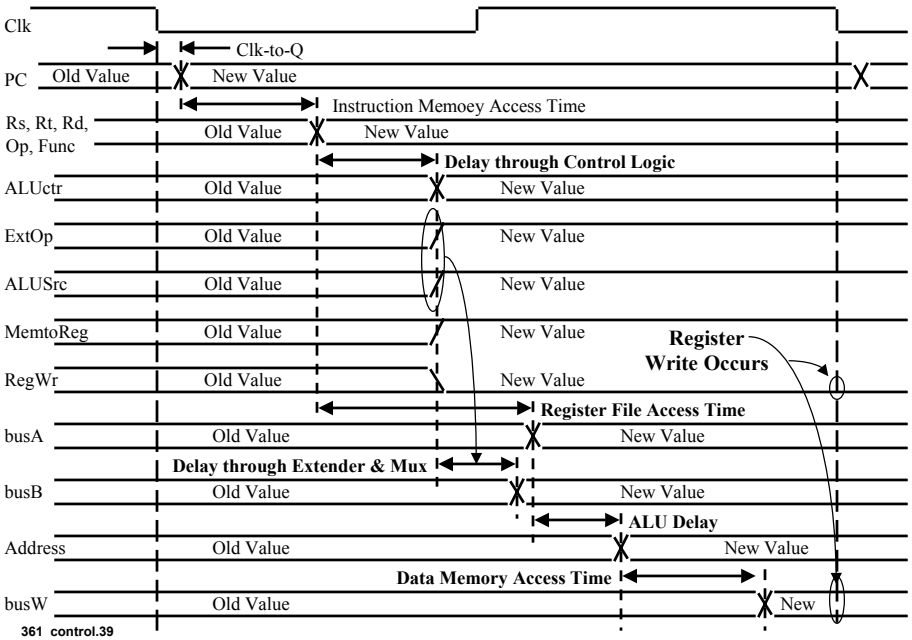


° **All storage elements are clocked by the same clock edge**

° **Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew**

° **(CLK-to-Q + Shortest Delay Path - Clock Skew) > Hold Time**

Worst Case Timing (Load)



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

° Single cycle datapath => CPI=1, CCT => long

° 5 steps to design a processor

- 1. Analyze instruction set => datapath requirements
- 2. Select set of datapath components & establish clock methodology
- 3. Assemble datapath meeting the requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- 5. Assemble the control logic

° Control is the hard part

° MIPS makes control easier

- Instructions same size
- Source registers always in same place
- Immediates same size, location
- Operations always on registers/immediates