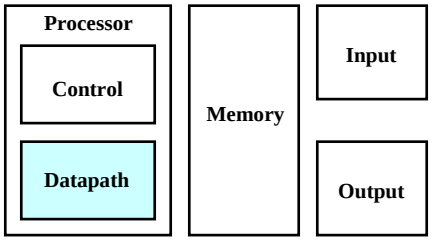
CS420/520
Computer Architecture I

Single Cycle Datapath & Control

Dr. Xiaobo Zhou
Department of Computer Science

The Big Picture: Where are We Now?

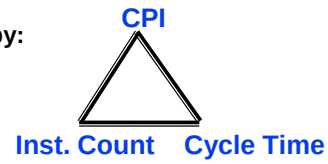
- The Five Classic Components of a Computer



- Today's Topic: Datapath Design

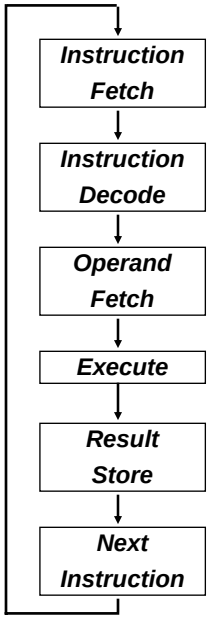
The Big Picture: The Performance Perspective

- Performance of a machine was determined by:
  - Instruction count
  - Clock cycle time
  - Clock cycles per instruction



- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- In the next two lectures:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time

Review: Execution Cycle



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in storage (Register/Memory) for later use

Determine successor instruction

Datapath Design Procedure

° 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

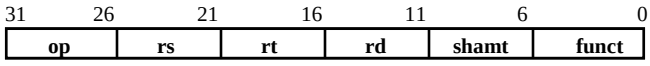
° MIPS makes it easier
- Instructions same size/length
- Source registers always in same place
- Immediates same size, location
- Operations always on registers/immediates

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

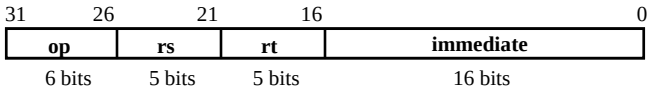
The MIPS Instruction Formats

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

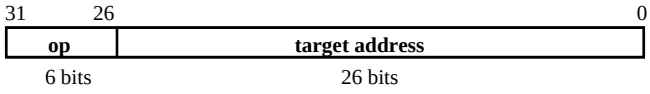
- R-type



- I-type



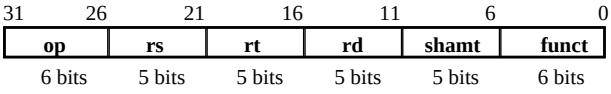
- J-type



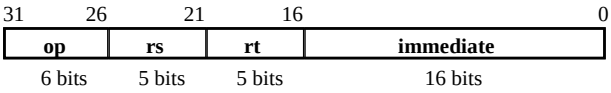
° The different fields are:
- op: operation of the instruction
- rs, rt, rd: the source and destination register specifiers
- shamt: shift amount
- funct: selects the variant of the operation in the “op” field
- address / immediate: address offset or immediate value
- target address: target address of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

The MIPS Subset

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



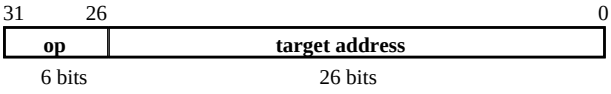
° OR Immediate:
  - ori rt, rs, imm16



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

° BRANCH:
  - beq rs, rt, imm16

° JUMP:
  - j target



The Design is to Represent: ALU – Central Part of CPU

(1) Functional Specification

Inputs: 2 x 32 bit operands- A, B; 1 bit carry input- Cin.

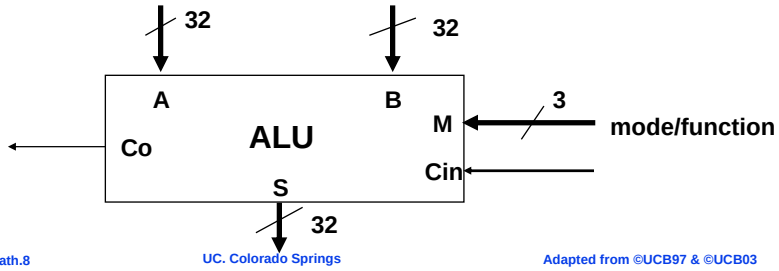
Outputs: 1 x 32 bit result- S; 1 bit carry output- Co.

Operations: ADD (A plus B plus Cin), SUB (A minus B minus Cin), AND, OR, XOR

Performance: left unspecified for now!

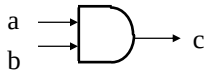
(2) Block Diagram

Understand the data and control flows



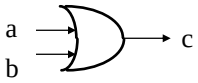
4 Hardware Building Blocks

° **AND gate (c = a & b)**



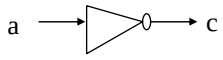
a	b	c = a & b
0	0	0
1	0	0
0	1	0
1	1	1

° **OR gate (c = a | b)**



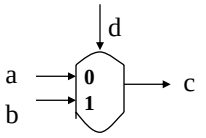
a	b	c = a | b
0	0	0
1	0	1
0	1	1
1	1	1

° **Inverter (c = !a)**



a	c = ! A
0	1
1	0

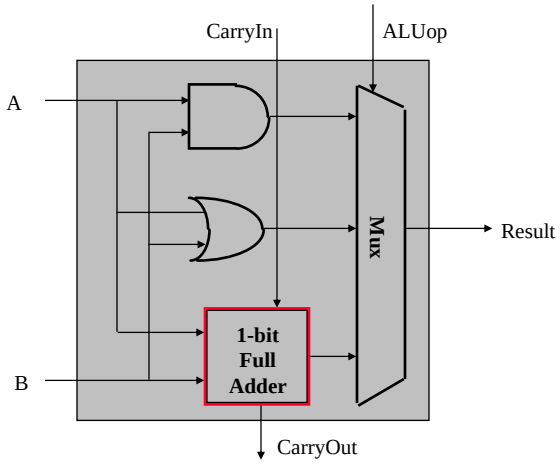
° **Multiplexor**
if d==0, c=a;
otherwise c= b



d	c
0	a
1	b

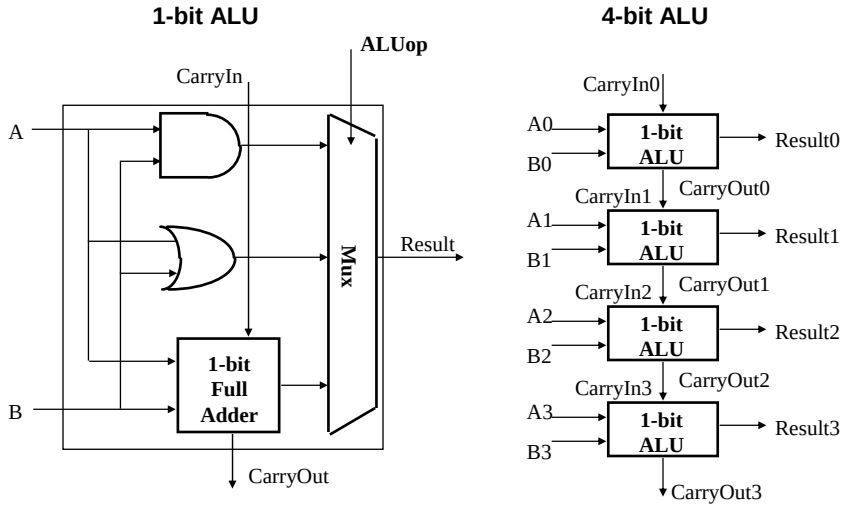
A One Bit ALU

° **This 1-bit ALU will perform AND, OR, and ADD**



A 1-bit ALU and a 4-bit ALU

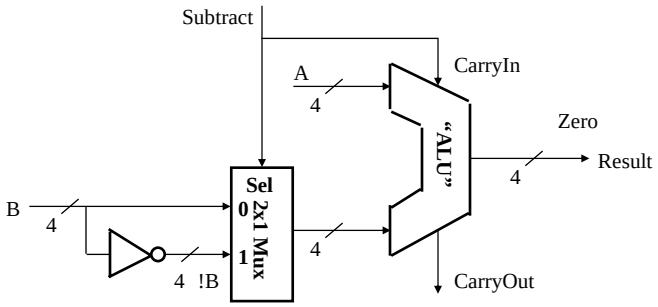
° This 1-bit ALU will perform AND, OR, and ADD



How About Subtraction?

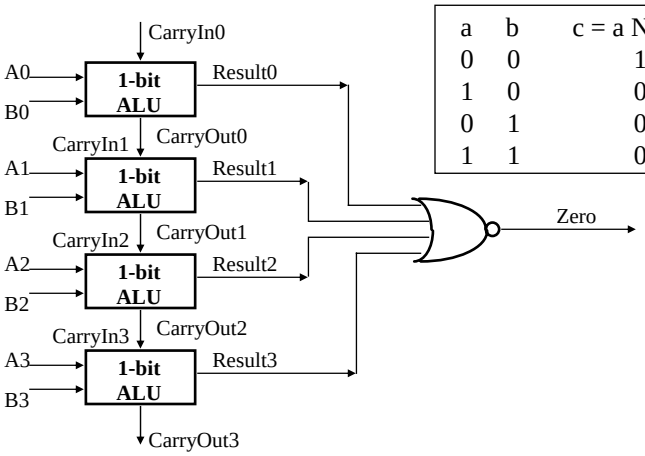
° Keep in mind the followings:
- (A - B) is the that as: A + (-B)
- 2's Complement: Take the inverse of every bit and add 1

° Bit-wise inverse of B is !B:
- A + !B + 1 = A + (!B + 1) = A + (-B) = A - B



Zero Detection Logic

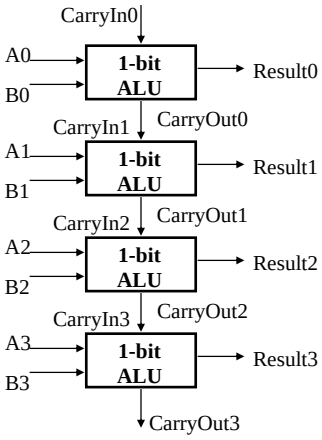
- A = B is the same as A - B = 0
- Zero Detection Logic is just a one BIG NOR gate
  - Any non-zero input to the NOR gate will cause its output to be zero

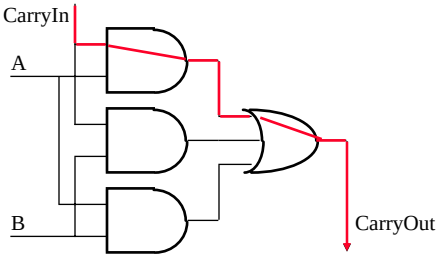


a	b	c = a NOR b
0	0	1
1	0	0
0	1	0
1	1	0

The Disadvantage of Ripple Carry

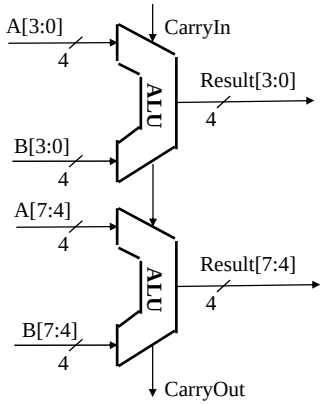
- The adder we just built is called a “Ripple Carry Adder”
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for a N-bit adder: 2N-gate delay





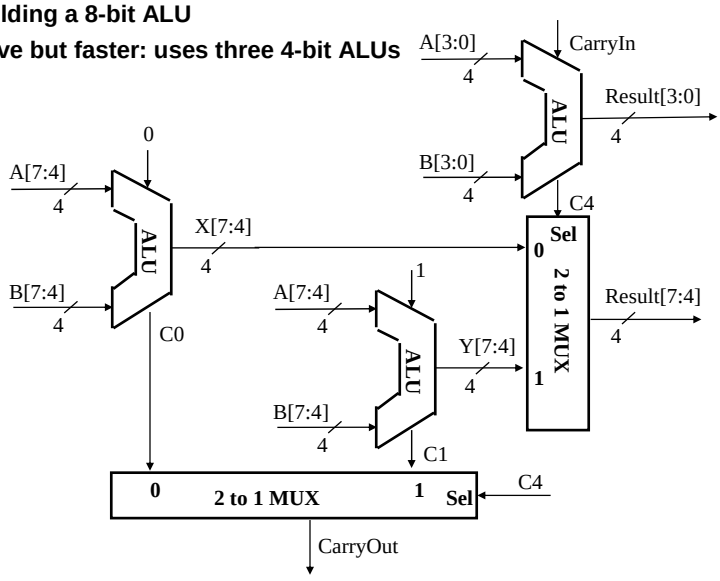
Carry Select Header

- Consider building a 8-bit ALU
  - Simple: connects two 4-bit ALUs in series (ripple carry)



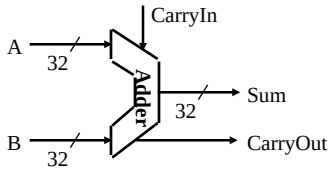
Carry Select Header (Continue)

- Consider building a 8-bit ALU
  - Expensive but faster: uses three 4-bit ALUs

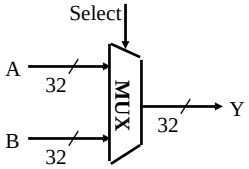


Combinational Logic Elements

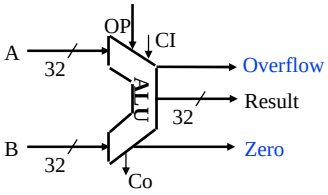
° **Adder**



° **MUX**

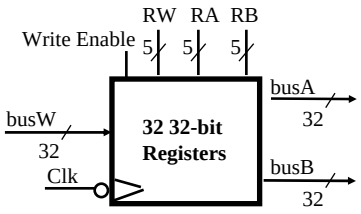


° **ALU**



Storage Element: Register File

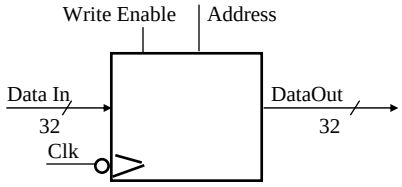
° **Register File consists of 32 registers:**
- **Two 32-bit output busses: busA and busB**
- **One 32-bit input bus: busW**



° **Register is selected by:**
- **RA selects the register to put on busA**
- **RB selects the register to put on busB**
- **RW selects the register to be written via busW when Write Enable is 1**

° **Clock input (CLK)**
- **The CLK input is a factor ONLY during write operation**
- **During read operation, RF behaves as a combinational logic block:**
  - **RA or RB valid => busA or busB valid after "access time."**

Storage Element: Idealized Memory

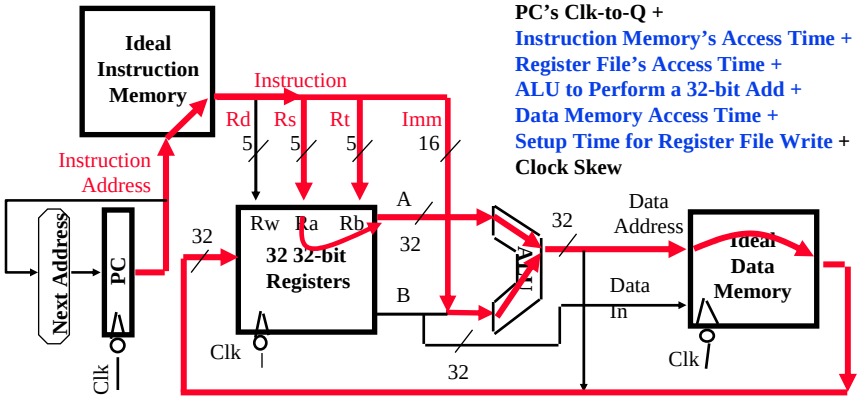


- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, IM behaves as a combinational logic block:
    - Address valid => Data Out valid after "access time."

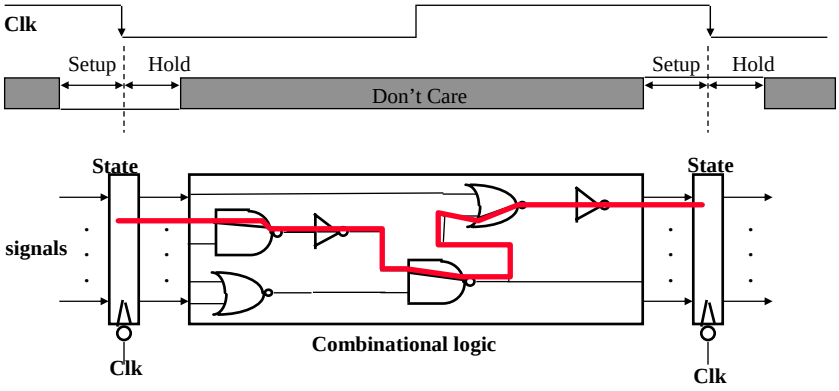
An Abstract View of the Critical Path

- 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."

Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew



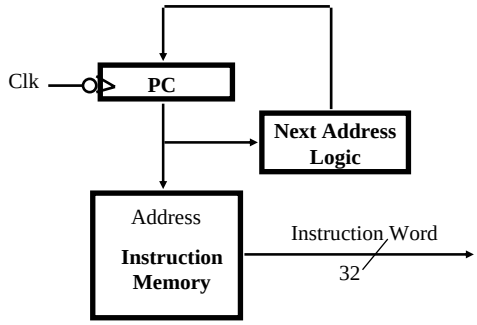
Clocking Methodology



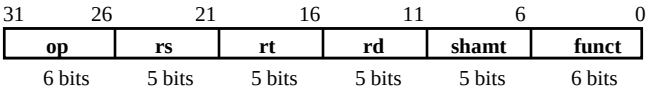
- A clocking methodology defines when signals can be read and written.
- For simplicity, we suppose an edge-triggered clocking methodology.
- All storage elements are clocked by the same clock edge
  - Edge-triggered: all stored values are updated on a clock edge

Overview of the Instruction Fetch Unit

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



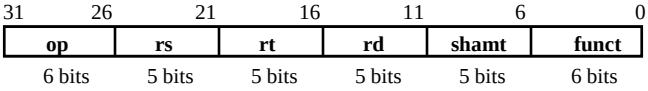
The ADD Instruction



° **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**

The Subtract Instruction



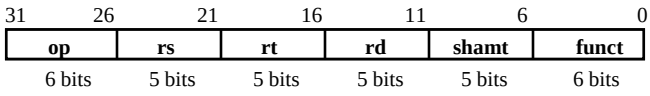
° **sub rd, rs, rt**

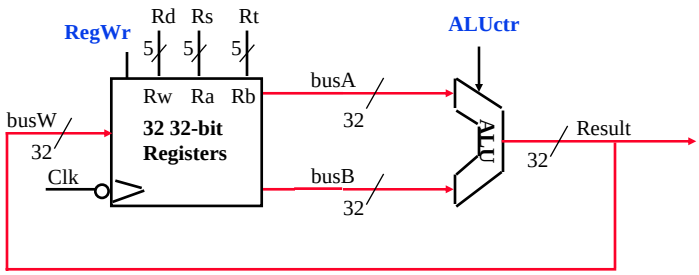
- **mem[PC]** **Fetch the instruction from memory**
- **R[rd] <- R[rs] - R[rt]** **The actual operation**
- **PC <- PC + 4** **Calculate the next instruction's address**

Datapath for Register-Register Operations

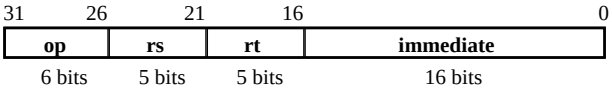
° R[rd] <- R[rs] op R[rt] Example: add rd, rs, rt

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



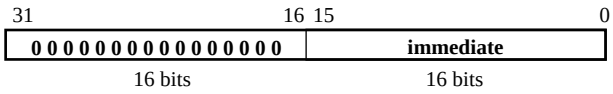


The OR Immediate Instruction



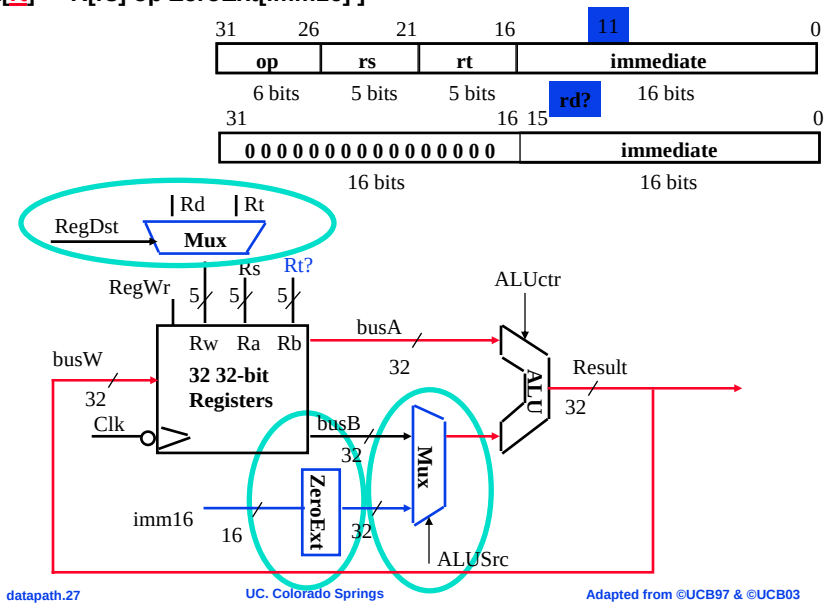
° ori rt, rs, imm16

- 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



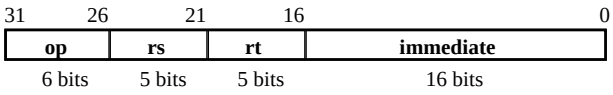
Datapath for Logical Operations with Immediate

° R[rt] <- R[rs] op ZeroExt[imm16]]

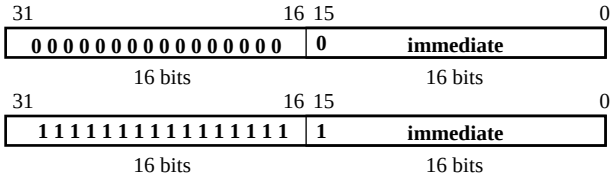


The Load Instruction

° lw rt, rs, imm16

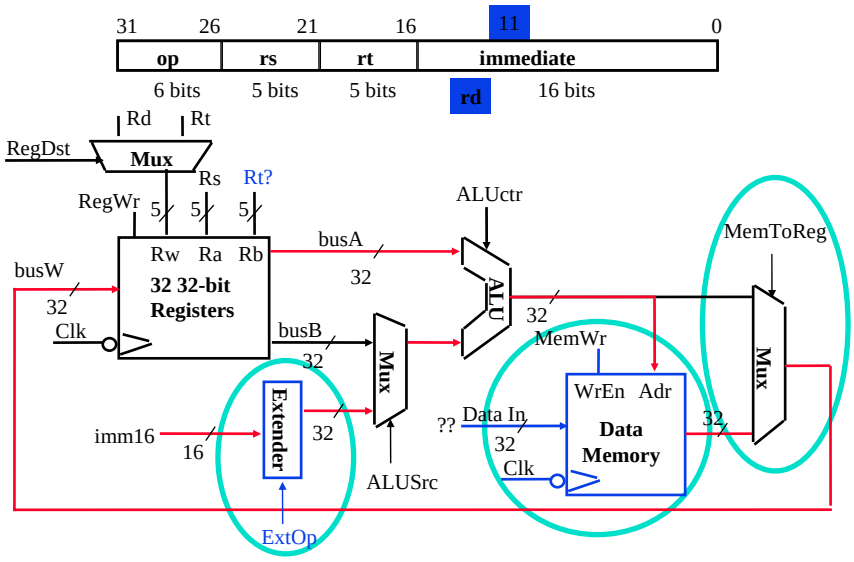


- mem[PC] Fetch the instruction from memory
- Addr <- R[rs] + SignExt(imm16)
  Calculate the memory address
  R[rt] <- Mem[Addr] Load the data into the register
- PC <- PC + 4 Calculate the next instruction's address

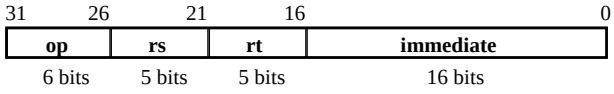


Datapath for Load Operations

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



The Store Instruction

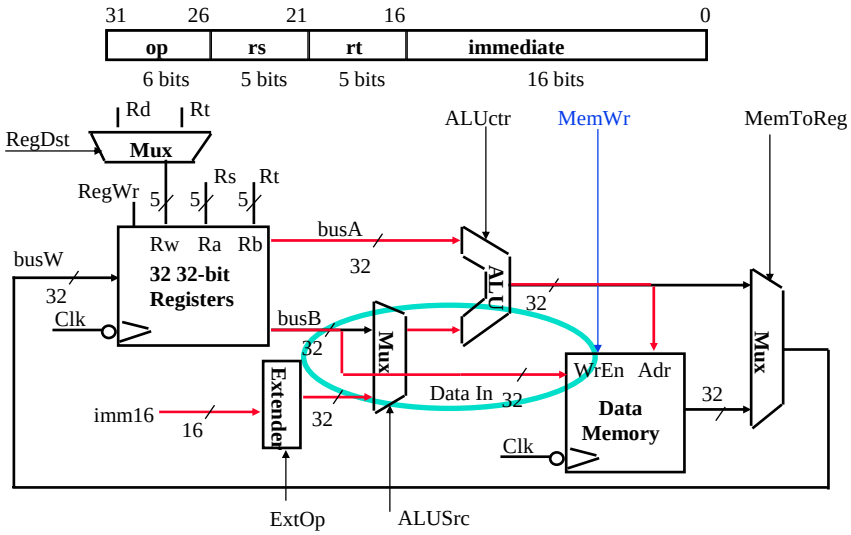


° sw rt, rs, imm16

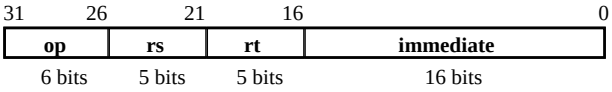
- mem[PC] Fetch the instruction from memory
- Addr <- R[rs] + SignExt(imm16)
  Calculate the memory address
- Mem[Addr] <- R[rt] Store the register into memory
- PC <- PC + 4 Calculate the next instruction's address

Datapath for Store Operations

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



The Branch Instruction

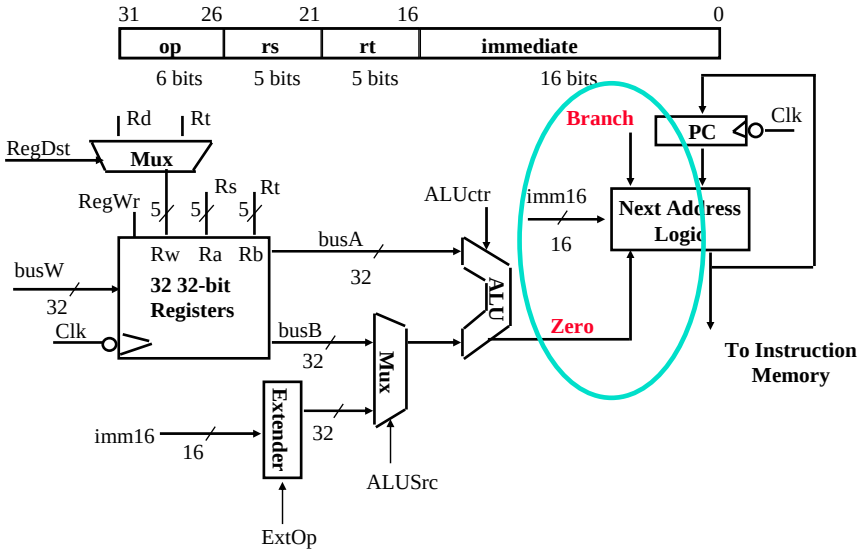


° beq rs, rt, imm16

- 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 <- 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!



Binary Arithmetics 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

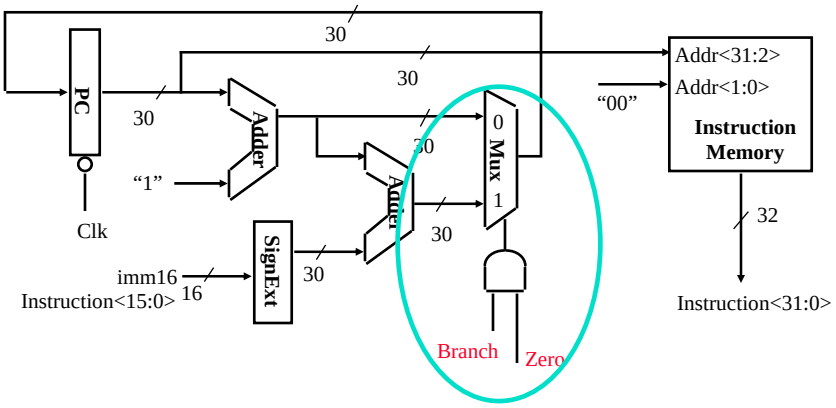
° The magic number “4” always comes up because:
- The 32-bit PC is a byte address
- And all our instructions are 4 bytes (32 bits) long

° 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”

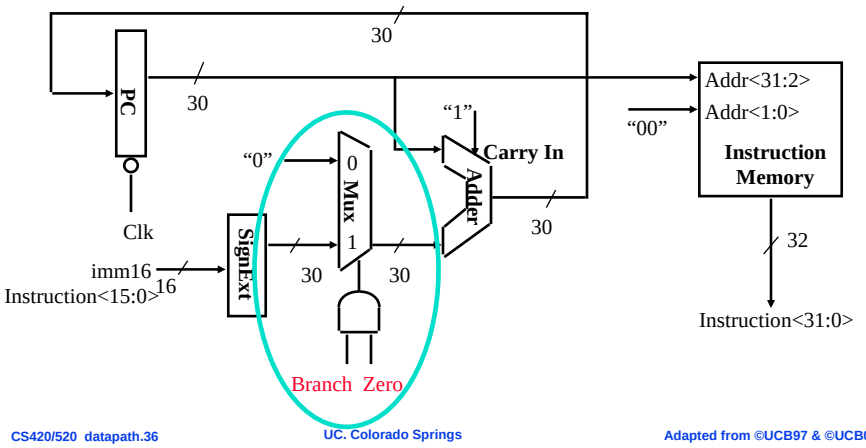
Next Address Logic: Expensive and Fast Solution

- 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"

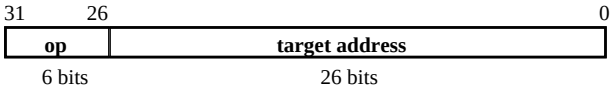


Next Address Logic: Cheap and Slow Solution

- Why is this slow?
  - Cannot start the address add until Zero (output of ALU) is valid
- Does it matter that this is slow in the overall scheme of things?
  - Probably not here. Critical path is the load operation.



The Jump Instruction



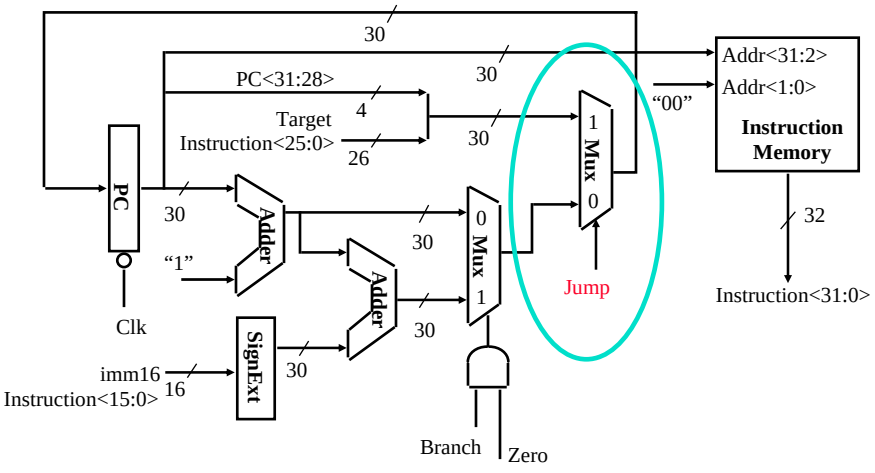
° **j target**

- **mem[PC]** **Fetch the instruction from memory**
- **PC<31:2> <- PC<31:28> concat target<25:0>**
  Calculate the next instruction's address

Instruction Fetch Unit

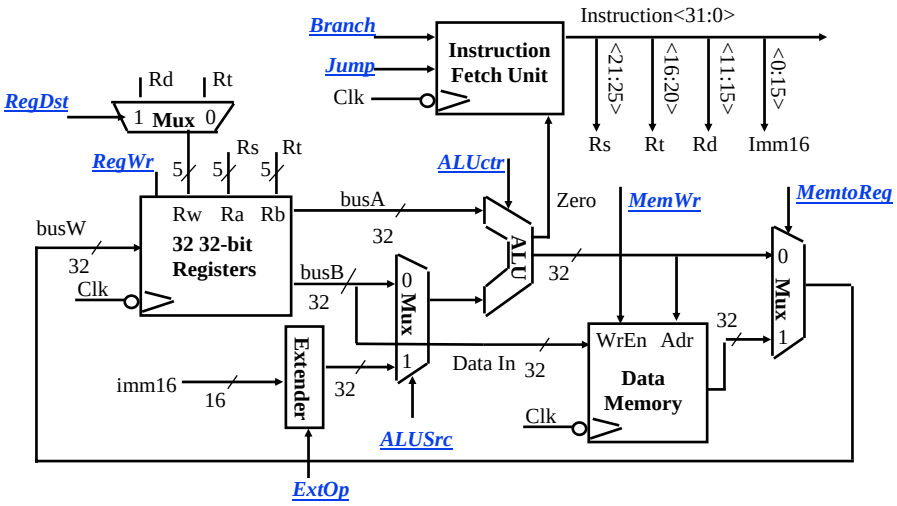
° **j target**

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



Putting it All Together: A Single Cycle Datapath

° We have everything except control signals (underline), to be continued ...



Summary of Datapath Design

° 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

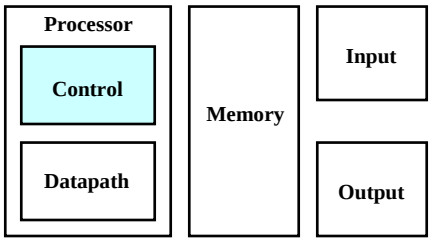
° MIPS makes it easier
- Instructions same size
- Source registers always in same place
- Immediates same size, location
- Operations always on registers/immediates

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

° What is next?: implementing control

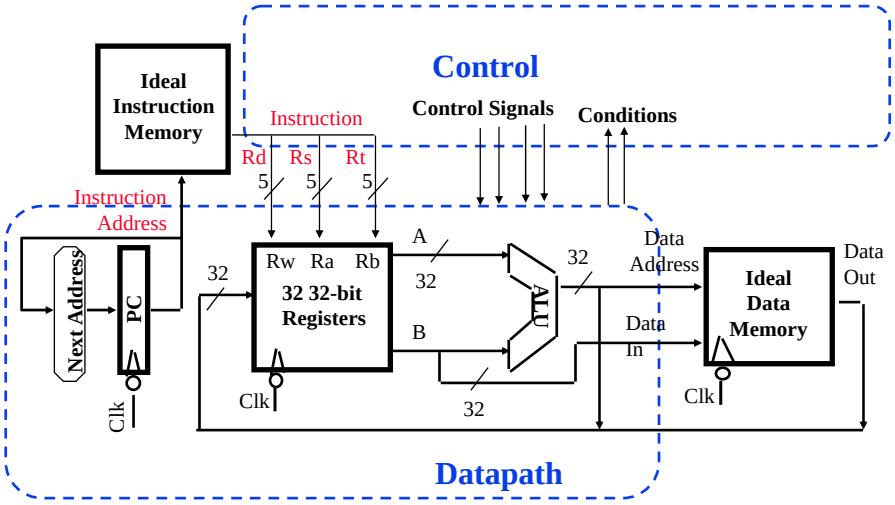
The Big Picture: Where are We Now?

° The Five Classic Components of a Computer

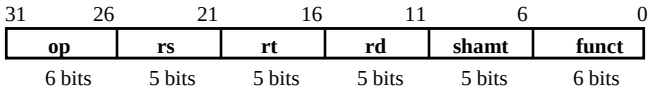


° Topic: Designing the Control for the Single Cycle Datapath

An Abstract View of the Control Implementation



The ADD Instruction

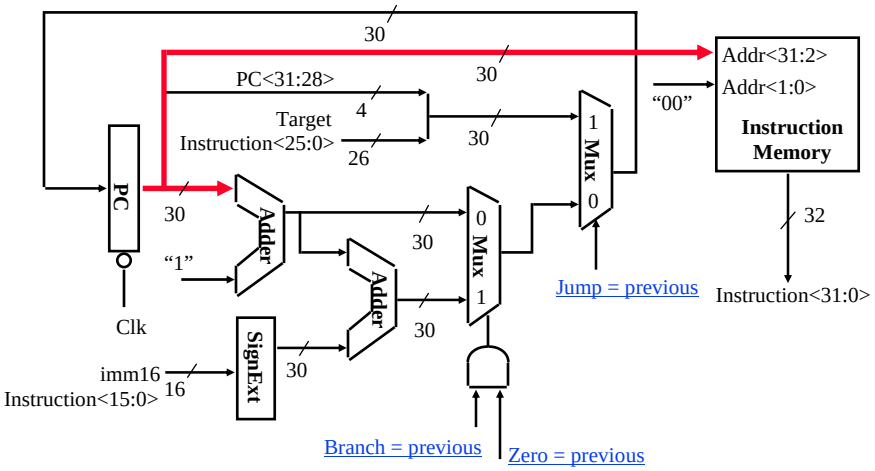


° **add rd, rs, rt**

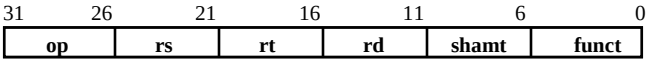
- **mem[PC]** — **Fetch the instruction from memory**
- **R[rd] <- R[rs] + R[rt]** — **The actual operation**
- **PC <- PC + 4** — **Calculate the next instruction's address**

Instruction Fetch Unit at the Beginning of Add / Subtract

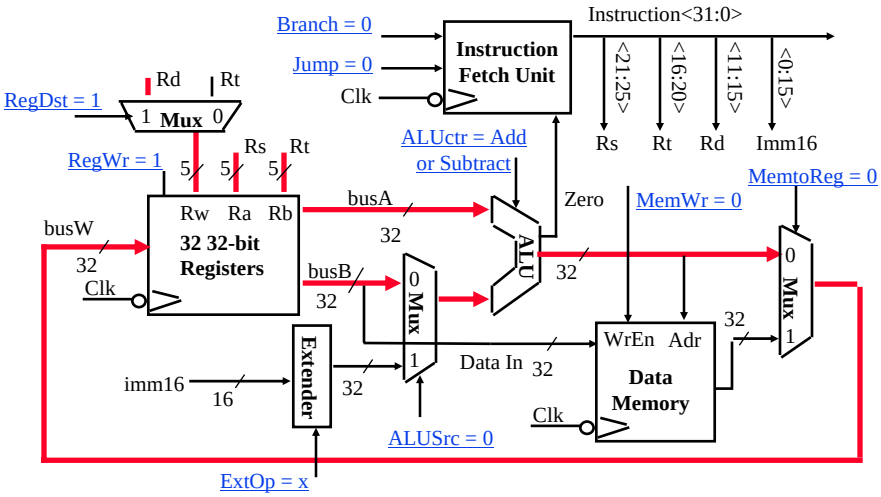
° **Fetch the instruction from Instruction memory: Instruction <- mem[PC]**
- **This is the same for all instructions**



The Single Cycle Datapath during Add and Subtract



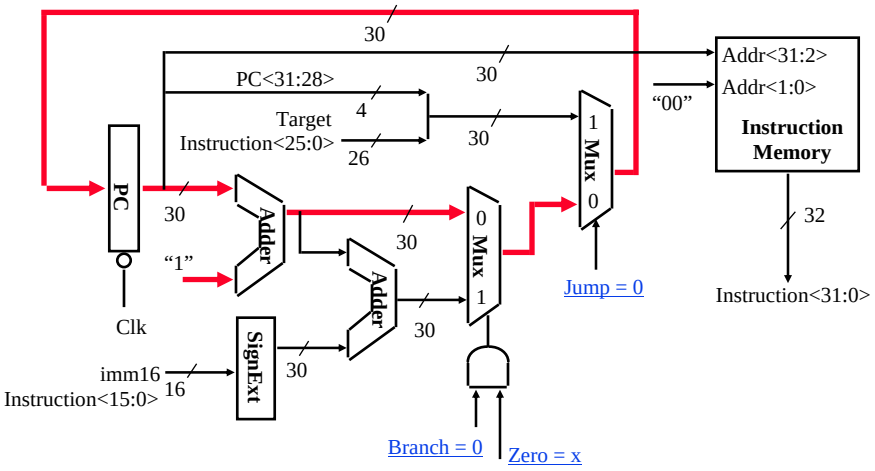
° R[rd] <- R[rs] + / - R[rt]



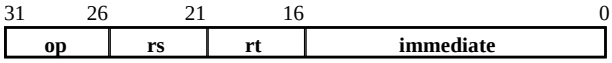
Instruction Fetch Unit at the End of Add and Subtract

° PC <- PC + 4

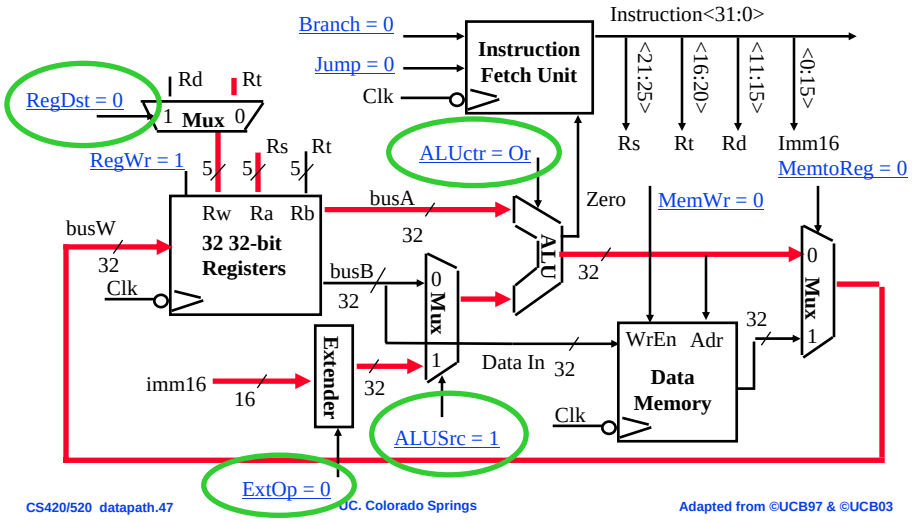
- This is the same for all instructions except: Branch and Jump



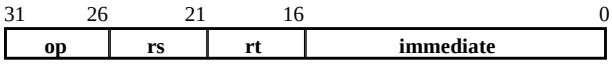
The Single Cycle Datapath during Or Immediate



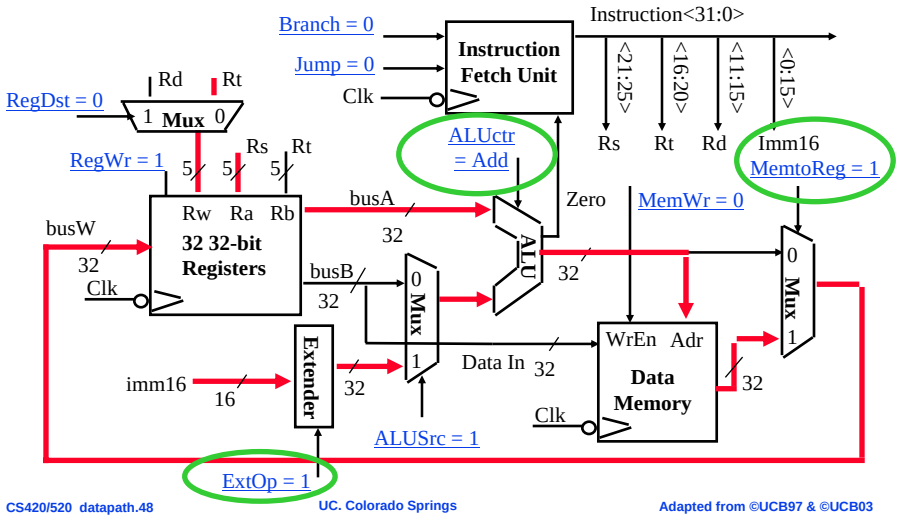
° R[rt] <- R[rs] or ZeroExt[Imm16]



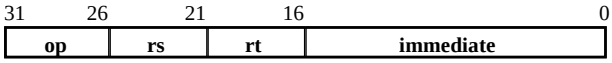
The Single Cycle Datapath during Load



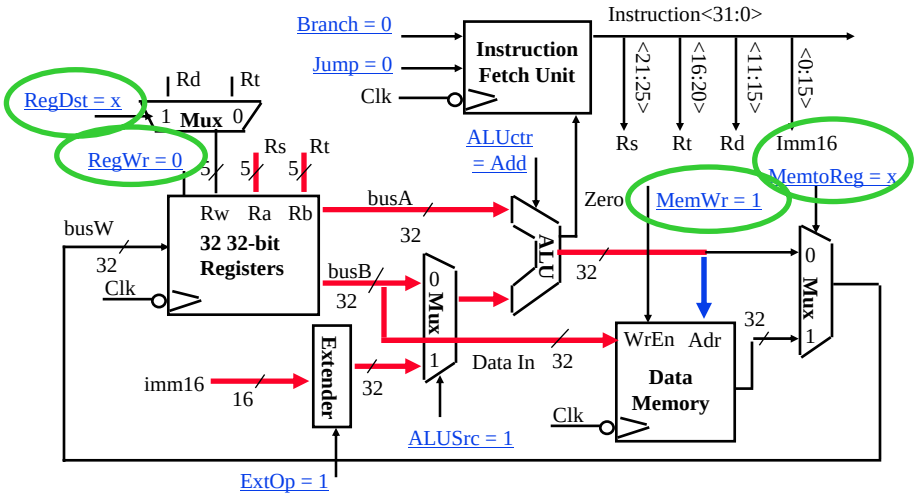
° R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



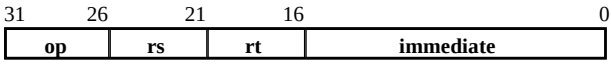
The Single Cycle Datapath during Store



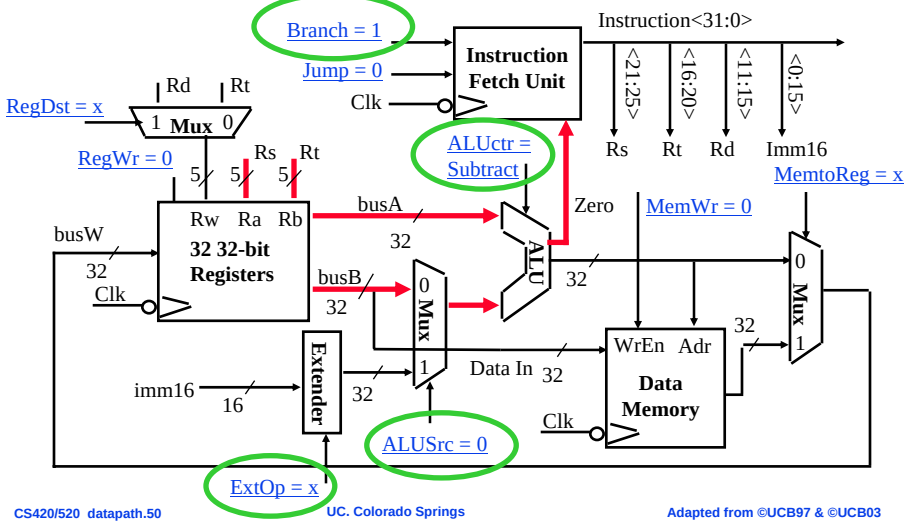
° Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



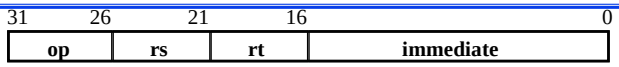
The Single Cycle Datapath during Branch



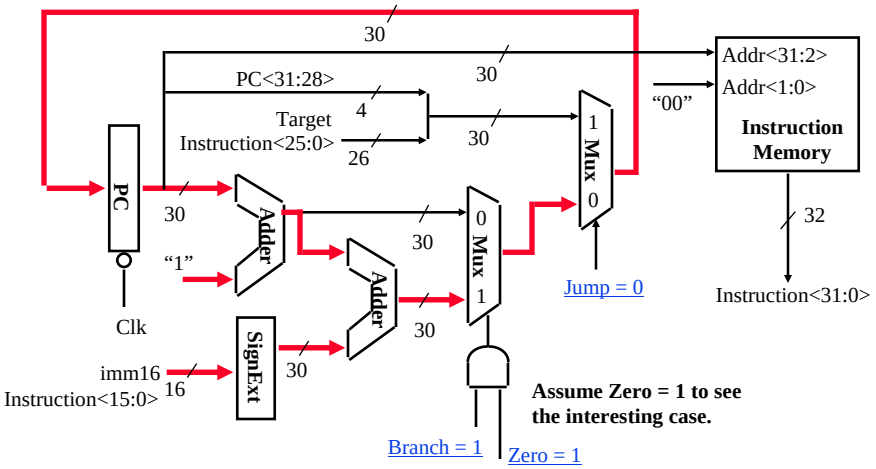
° if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



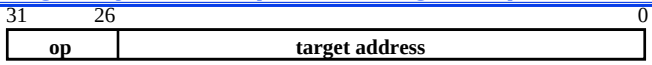
Instruction Fetch Unit at the End of Branch



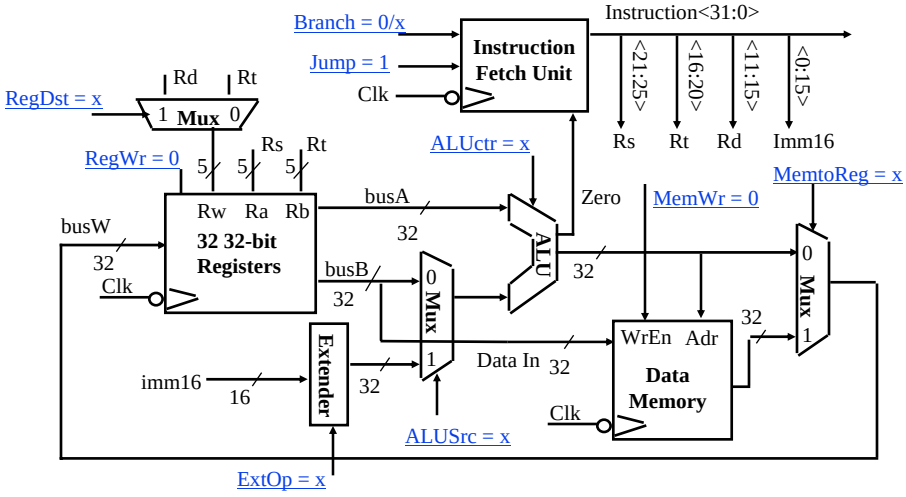
° **if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4**



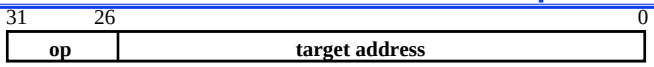
The Single Cycle Datapath during Jump



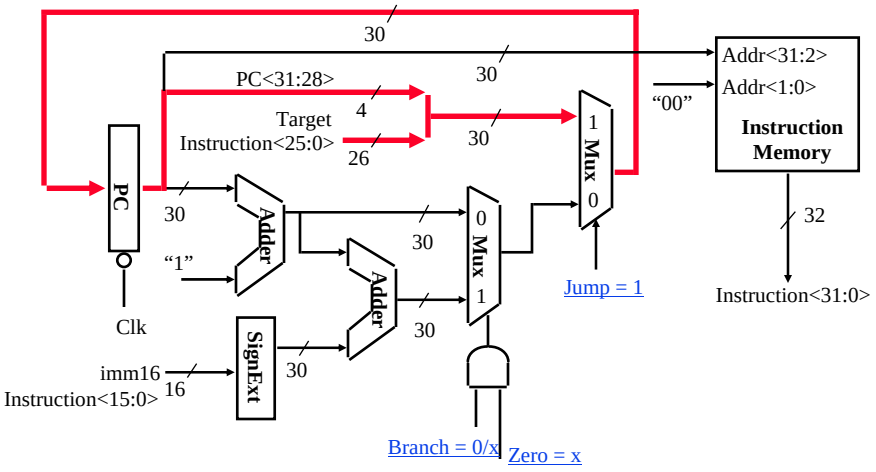
° **Nothing to do! Make sure control signals are set correctly!**



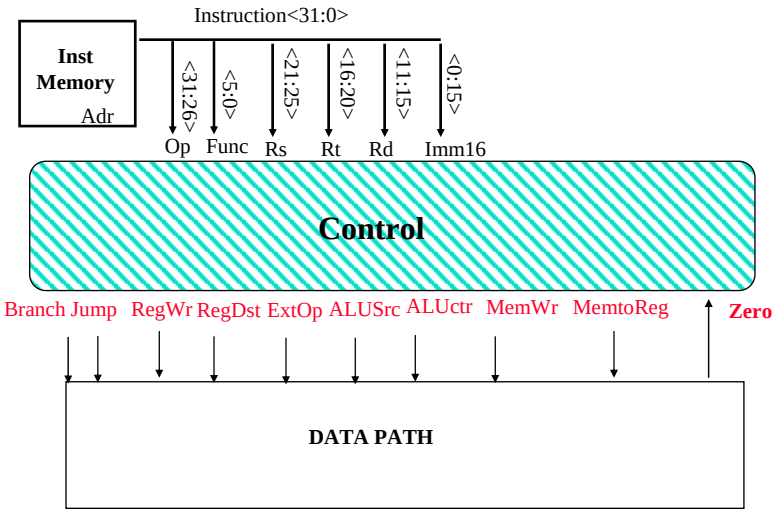
Instruction Fetch Unit at the End of Jump



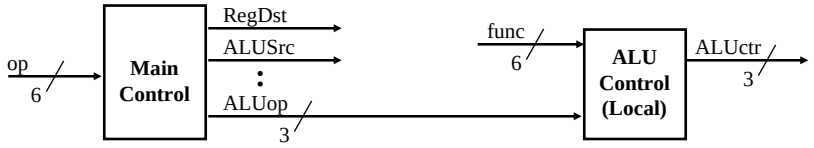
° PC <- PC<31:28> concat target<25:0> concat “00”



Step 4: Given Datapath: ---> Control



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

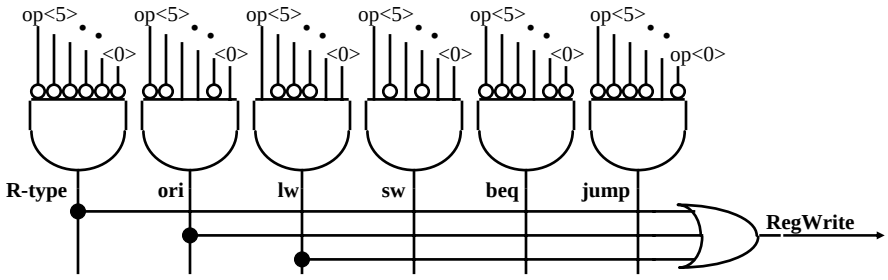
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

° **RegWrite = R-type + ori + lw**

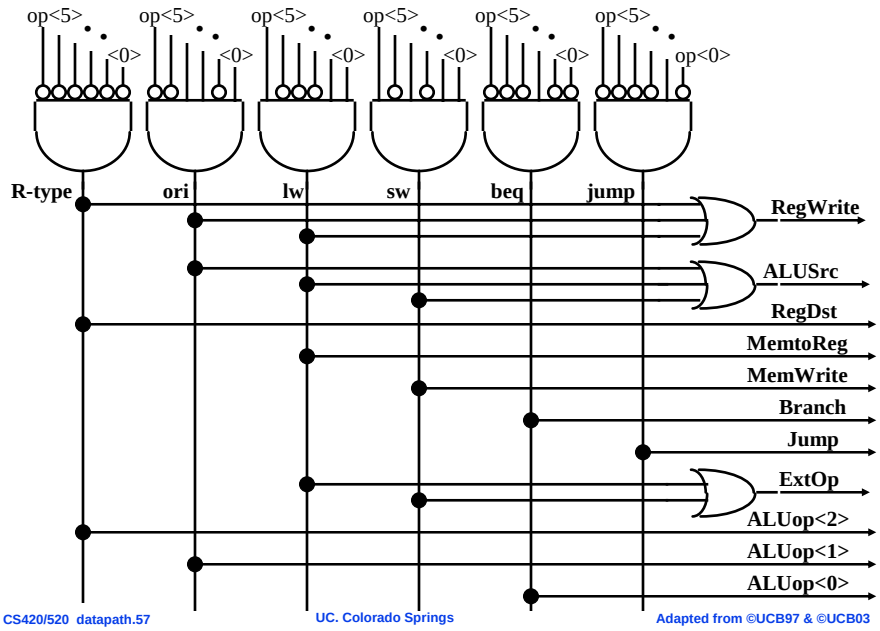
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> **(R-type)**

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

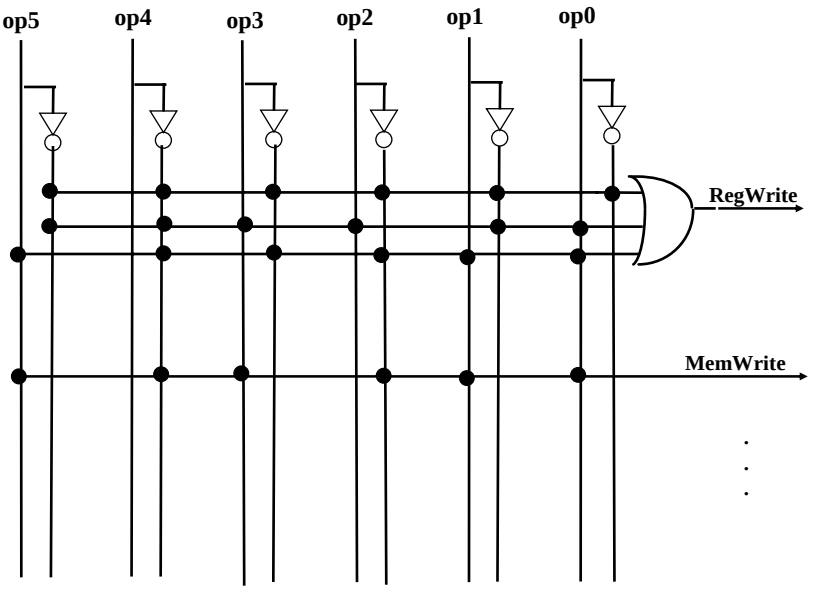
+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> **(lw)**



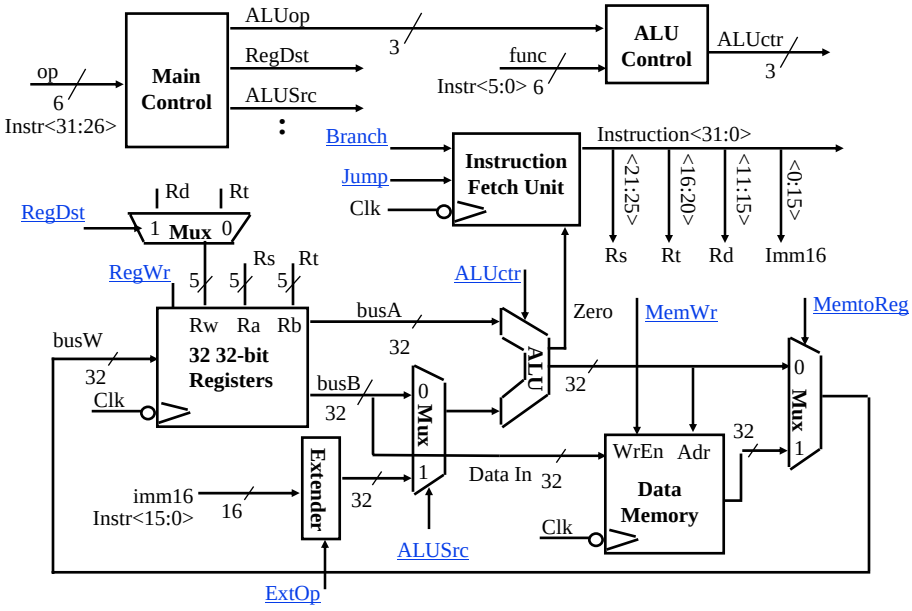
PLA Implementation of the Main Control



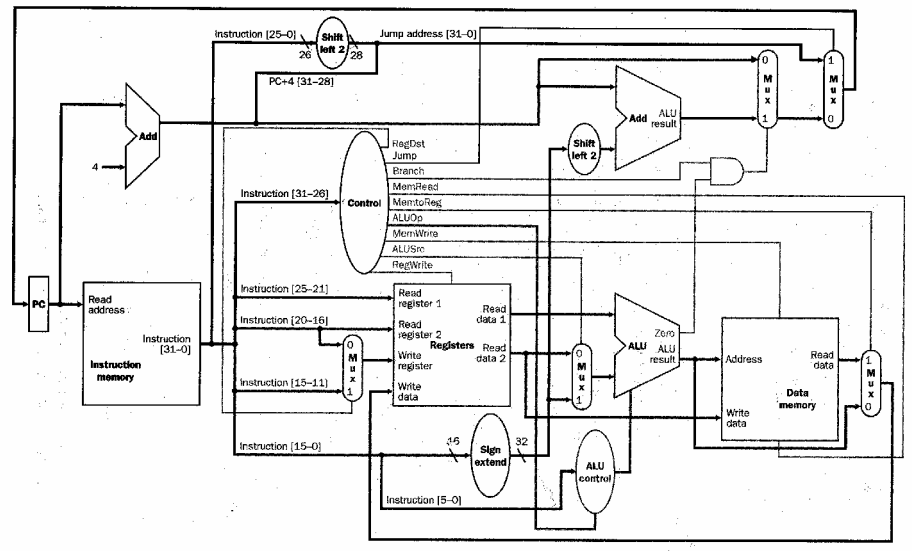
PLA Implementation of the Main Control: Drawing II



Putting it All Together: A Single Cycle Processor



Putting it All Together: An Extended View



Where to get more information?

- CO2: Chapter 4.1 to 4.5 (pp.210 – 236); Chapter 5.1 to 5.3

  CO3: Chapter 3.1 – 3.3 (pp. 160 – 176); Chapter 5.1 to 5.4
  - David Patterson and John Hennessy, "Computer Organization & Design: The Hardware / Software Interface," Morgan Kaufman Publishers; CO2 (2nd edition) and CO3 (3rd edition)

- One of the best PhD thesis on processor design:
  - Manolis Katevenis, "Reduced Instruction Set Computer Architecture for VLSI," PhD Dissertation, EECS, U.C. Berkeley, 1982.

- For a reference on the MIPS architecture:
  - Gerry Kane, "MIPS RISC Architecture," Prentice Hall.

CO review: Two's Complement Representation

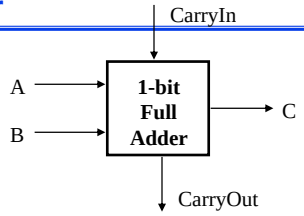
- 2's complement representation of negative numbers (signed)
  - Bitwise inverse and add 1
  - The MSB is always "1" for negative number => sign bit

- Biggest 4-bit Binary Number: 7 Smallest 4-bit Binary Number: -8

Decimal	Binary	Decimal	Bitwise Inverse (1's Com.)	2's Complement
0	0000	0	1111	0000
1	0001	-1	1110	1111
2	0010	-2	1101	1110
3	0011	-3	1100	1101
4	0100	-4	1011	1100
5	0101	-5	1010	1011
6	0110	-6	1001	1010
7	0111	-7	1000	1001
8	1000	-8	0111	1000

"Illegal" Positive Number!

CO review: A One-bit Full Adder

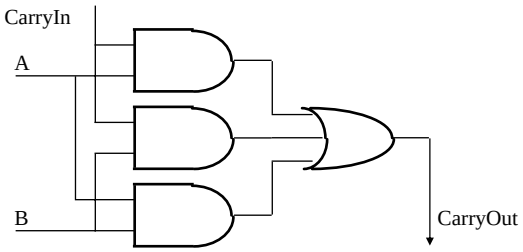


- This is also called a (3, 2) adder
- Half Adder: No CarryIn nor CarryOut
- Truth Table:

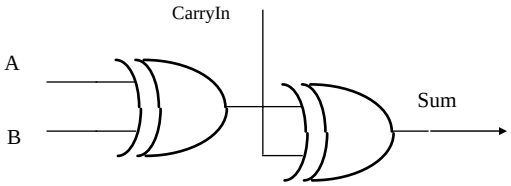
Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	0 + 0 + 0 = 00
0	0	1	0	1	0 + 0 + 1 = 01
0	1	0	0	1	0 + 1 + 0 = 01
0	1	1	1	0	0 + 1 + 1 = 10
1	0	0	0	1	1 + 0 + 0 = 01
1	0	1	1	0	1 + 0 + 1 = 10
1	1	0	1	0	1 + 1 + 0 = 10
1	1	1	1	1	1 + 1 + 1 = 11

CO review: Logic Diagrams for CarryOut

- CarryOut = (B & CarryIn) | (A & CarryIn) | (A & B)

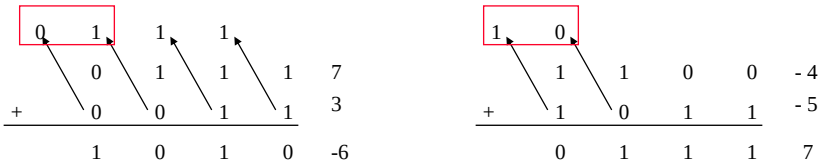


- Sum = A XOR B XOR CarryIn



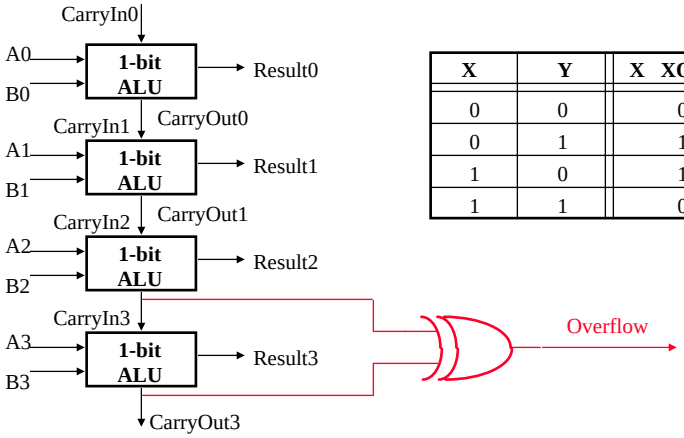
CO review: Overflow Detection

- Overflow: the result is too large (or too small) to represent properly
  - Example: - 8 < = 4-bit binary number <= 7
- When adding operands with different signs, overflow cannot occur!
- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive
- Optional homework exercise: Prove you can detect overflow by:
  - Carry into MSB ! = Carry out of MSB



CO review: Overflow Detection Logic

- Carry into MSB ! = Carry out of MSB
  - For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0