# CS152
# Computer Architecture and Engineering
# Lecture 7

## Designing a Single Cycle Datapath

February 18, 2004

John Kubiatowicz (www.cs.berkeley.edu/~kubitron)

lecture slides: http://inst.eecs.berkeley.edu/~cs152/



## Review: Sequential Logic + (Non-)blocking assignments

**Must be careful mixing zero-time blocking assignments and edge-triggering: Probably won't do what you expect when connecting it to other things!**

```
module FF (CLK,Q,D);
   input D, CLK;
   output Q; reg Q;
   always @ (posedge CLK)
      Q <= D;
endmodule // FF
```

**Good: Doesn't output until "after edge"**

```
module FF (CLK,Q,D);
   input D, CLK;
   output Q; reg Q;
   always @ (posedge CLK)
      Q = #5 D;
endmodule // FF
```

**Good: Outputs 5 units "after edge"**

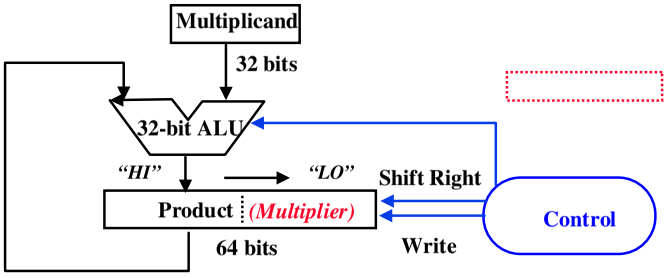
**Probably Not what you Expect:**
- **Hold time of 5 units**
- **glitches < 5 units ignored**

```
module FF (CLK,Q,D);
   input D, CLK;
   output Q; reg Q;
   always @ (posedge CLK)
      #5 Q = D;
endmodule // FF
```



## Review: MULTIPLY HARDWARE Version 3

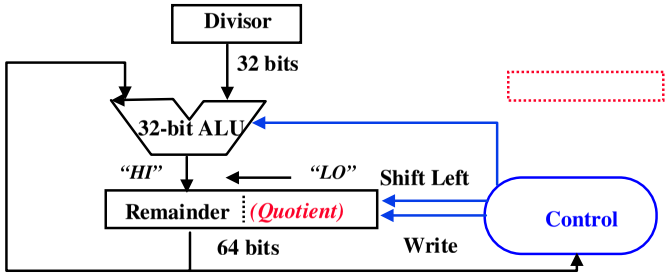
° 32-bit Multiplicand reg, 32-bit ALU,
64-bit Product reg (shift right), (0-bit Multiplier reg)





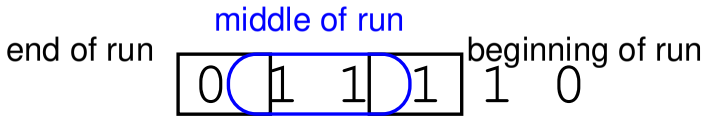

## Divide can use almost same hardware (From Book)

° 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg,
(0-bit Quotient reg)




° Multiplication and Division can use same hardware!

## Review: Booth's Algorithm⇒Alternate representation




| Current Bit | Bit to the Right | Explanation | Example | Op |
|---|---|---|---|---|
| 1 | 0 | Begins run of 1s | 000111**10**00 | sub $(\bar{1})$ |
| 1 | 1 | Middle of run of 1s | 00011**11**000 | none (0) |
| 0 | 1 | End of run of 1s | 000**1**111000 | add (1) |
| 0 | 0 | Middle of run of 0s | 00**0**1111000 | none (0) |

Examples (8 bits):

$-3 = 11111101 \Rightarrow 0000\bar{1}1\bar{1} = -4 + 2 - 1$

$14 = 00001110 \Rightarrow 0001001\bar{0} = 16 - 2$

$23 = 00010111 \Rightarrow 001\bar{1}100\bar{1} = 32 - 16 + 8 - 1$



## MIPS logical instructions

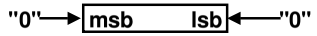
| Instruction | Example | Meaning | Comment |
|---|---|---|---|
| and | and $1,$2,$3 | $1 = $2 & $3 | 3 reg. operands; Logical AND |
| or | or $1,$2,$3 | $1 = $2 \| $3 | 3 reg. operands; Logical OR |
| xor | xor $1,$2,$3 | $1 = $2 ⊕ $3 | 3 reg. operands; Logical XOR |
| nor | nor $1,$2,$3 | $1 = ~($2 \|$3) | 3 reg. operands; Logical NOR |
| and immediate | andi $1,$2,10 | $1 = $2 & 10 | Logical AND reg, constant |
| or immediate | ori $1,$2,10 | $1 = $2 \| 10 | Logical OR reg, constant |
| xor immediate | xori $1, $2,10 | $1 = ~$2 &~10 | Logical XOR reg, constant |
| shift left logical | sll $1,$2,10 | $1 = $2 << 10 | Shift left by constant |
| shift right logical | srl $1,$2,10 | $1 = $2 >> 10 | Shift right by constant |
| shift right arithm. | sra $1,$2,10 | $1 = $2 >> 10 | Shift right (sign extend) |
| shift left logical | sllv $1,$2,$3 | $1 = $2 << $3 | Shift left by variable |
| shift right logical | srlv $1,$2, $3 | $1 = $2 >> $3 | Shift right by variable |
| shift right arithm. | srav $1,$2, $3 | $1 = $2 >> $3 | Shift right arith. by variable |



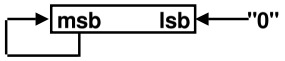
## Shifters

Two kinds:

*logical (RIGHT OR LEFT)*-- value shifted in is always "0"




*arithmetic*– (RIGHT ONLY), sign extend

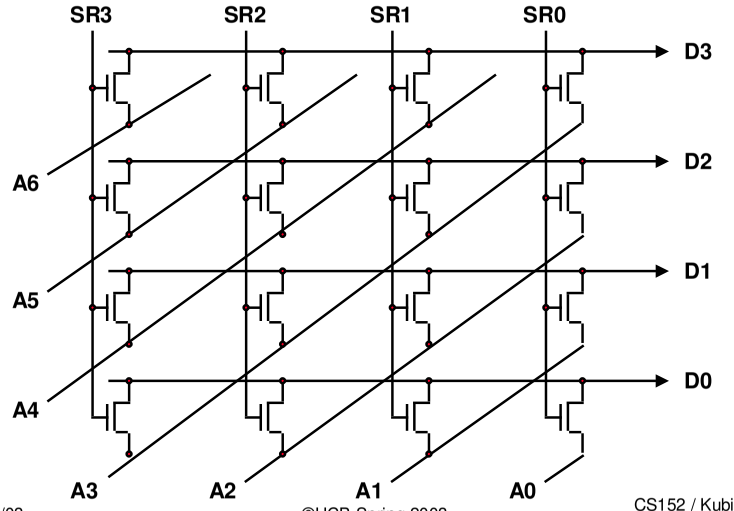



**Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!**



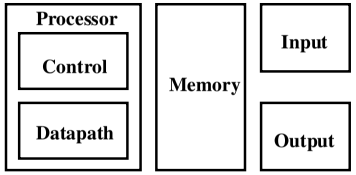
## Barrel Shifter

Technology-dependent solutions: transistor per switch

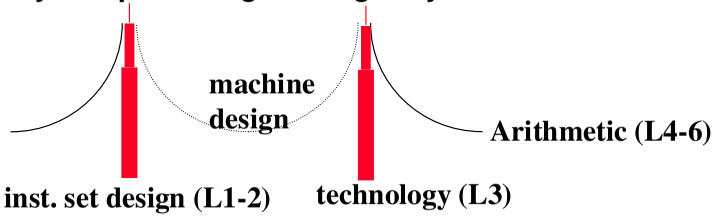


## The Big Picture: Where are We Now?

° The Five Classic Components of a Computer



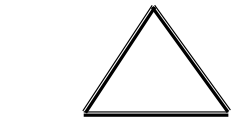

° Today's Topic: Design a Single Cycle Processor






## The Big Picture: The Performance Perspective

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




° Processor design (datapath and control) will determine:
- Clock cycle time
- Clock cycles per instruction

° Today:
- Single cycle processor:
  - Advantage: One clock cycle per instruction
  - Disadvantage: long cycle time



## How to Design a Processor: step-by-step

° 1. Analyze instruction set => datapath <u>requirements</u>
- the meaning of each instruction is given by the *register transfers*
- datapath must include storage element for ISA registers
  - possibly more
- datapath must support each register transfer

° 2. Select set of datapath components and establish clocking methodology

° 3. <u>Assemble</u> datapath meeting the requirements

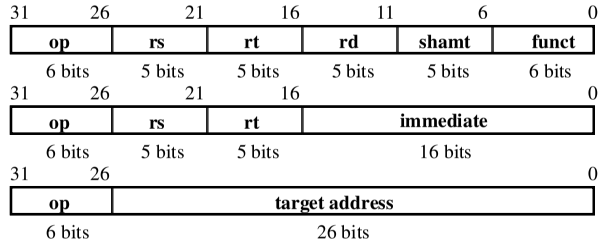
° 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.

° 5. Assemble the control logic



## The MIPS Instruction Formats

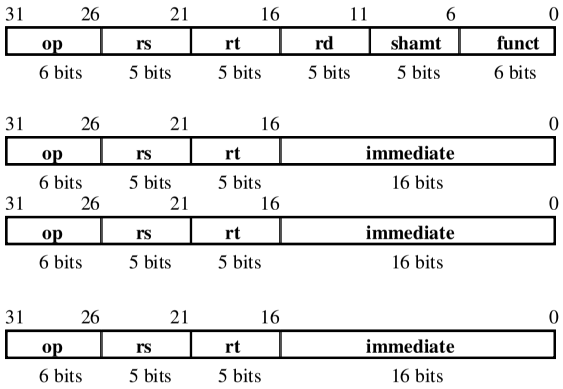
° All MIPS instructions are 32 bits long. The three instruction formats:




° 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

## Step 1a: The MIPS-lite Subset for today

- ADD and SUB
  - addU rd, rs, rt
  - subU rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE Word
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16






## Logical Register Transfers

- RTL gives the meaning of the instructions
- All start by fetching the instruction

op | rs | rt | rd | shamt | funct = MEM[ PC ]

op | rs | rt | Imm16 = MEM[ PC ]

| inst | Register Transfers | |
|---|---|---|
| ADDU | R[rd] <– R[rs] + R[rt]; | PC <– PC + 4 |
| SUBU | R[rd] <– R[rs] – R[rt]; | PC <– PC + 4 |
| ORi | R[rt] <– R[rs] \| zero_ext(Imm16); | PC <– PC + 4 |
| LOAD | R[rt] <– MEM[ R[rs] + sign_ext(Imm16)]; | PC <– PC + 4 |
| STORE | MEM[ R[rs] + sign_ext(Imm16) ] <– R[rt]; | PC <– PC + 4 |
| BEQ | if ( R[rs] == R[rt] ) then | PC <– PC + 4 + sign_ext(Imm16)] \|\| 00 |
| | else | PC <– PC + 4 |



## Step 1: Requirements of the Instruction Set

- Memory
  - instruction & data
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- PC
- Extender
- Add and Sub register or extended immediate
- Add 4 or extended immediate to PC

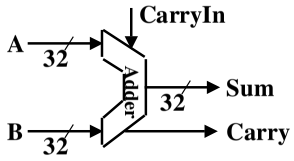


## Step 2: Components of the Datapath

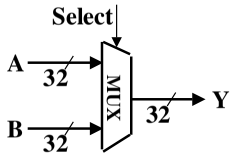
- Combinational Elements
- Storage Elements
  - Clocking methodology

## Combinational Logic Elements (Basic Building Blocks)

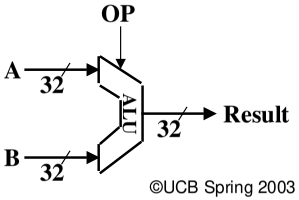
° Adder




° MUX




° ALU

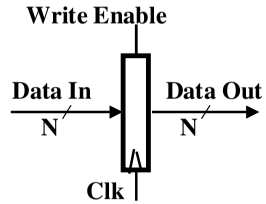





## Storage Element: Register (Basic Building Block)

° Register

- Similar to the D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable:
  - negated (0): Data Out will not change
  - asserted (1): Data Out will become Data In

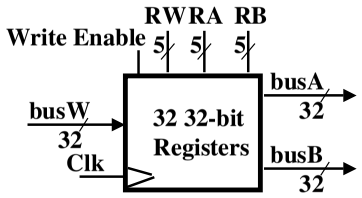





## Storage Element: Register File

° 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 (number) selects the register to put on busA (data)
- RB (number) selects the register to put on busB (data)
- RW (number) selects the register to be written via busW (data) when Write Enable is 1

° Clock input (CLK)

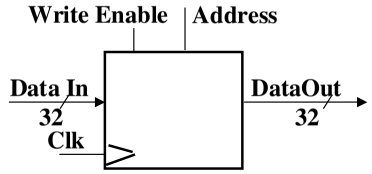
- The CLK input is a factor ONLY during write operation
- During read operation, behaves as a combinational logic block:
  - RA or RB valid => busA or busB valid after "access time."



## Storage Element: Idealized Memory

° Memory (idealized)

- One input bus: Data In
- One output bus: Data Out



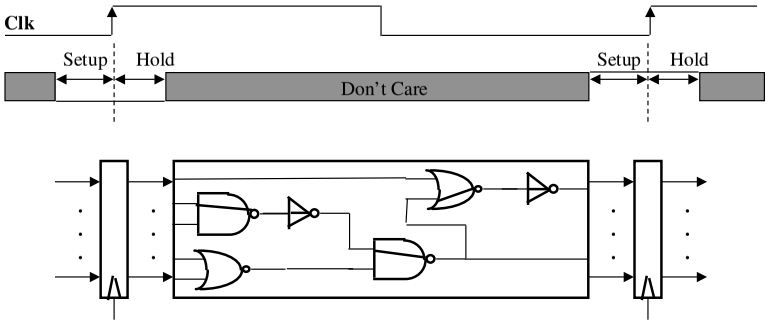

° Memory word is selected by:

- 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, behaves as a combinational logic block:
  - Address valid => Data Out valid after "access time."

## Recall: Clocking Methodology




° All storage elements are clocked by the same clock edge

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



## Administrivia

° LAB 2 due tomorrow at midnight
  • Must demonstrate your lab to TAs in section
° Tomorrow Section in LAB (119 Cory)
  • Must form up groups in tomorrow
    - 4 or 5 person groups
    - No 3 person groups (this means that not all 5-person groups may be able to exist)
° LAB 3/Homework 3 out tomorrow
  • Build a single-cycle processor in Hardware!
  • Start working in Groups
° Today's material in Chapter 5
° First Midterm: Wednesday, March 10
  • 5:30 – 8:30 in 306 Soda
  • La Val's Afterwards



## Step 3: Assemble DataPath meeting our requirements

° Register Transfer Requirements
  ⇒ Datapath Assembly

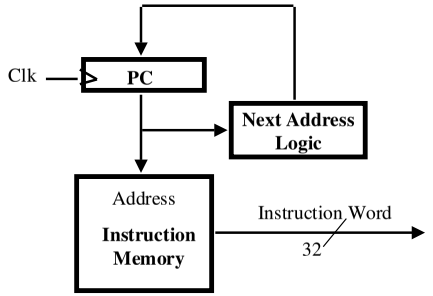
° Instruction Fetch

° Read Operands and Execute Operation



## 3a: Overview of the Instruction Fetch Unit

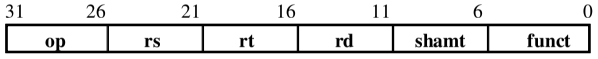
° The common RTL operations
  • Fetch the Instruction: mem[PC]
  • Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- "something else"

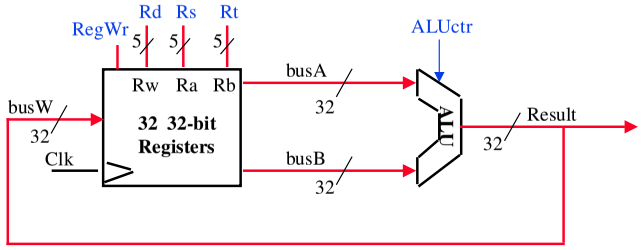


## 3b: Add & Subtract

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

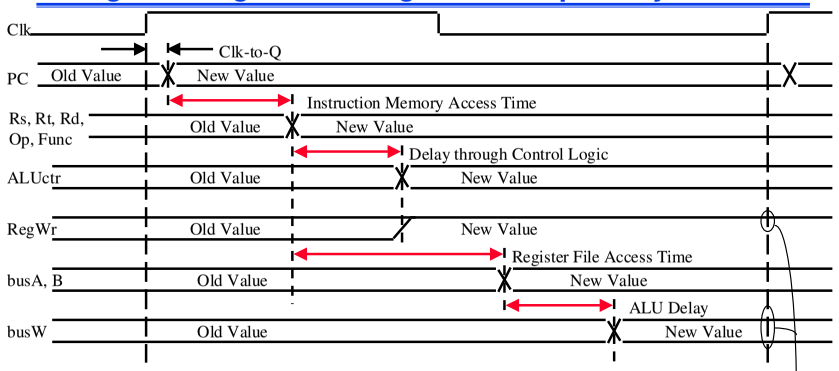
- Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
- ALUctr and RegWr: control logic after decoding the instruction

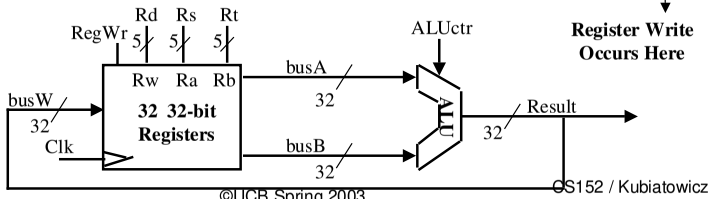





## Register-Register Timing: One complete cycle

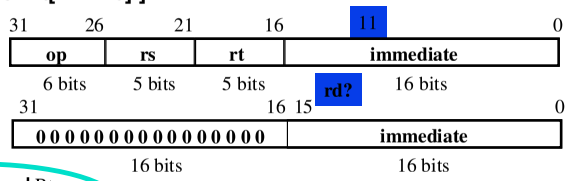


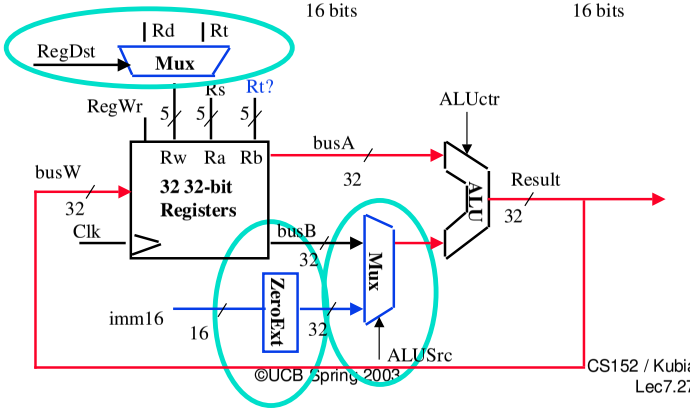






## 3c: Logical Operations with Immediate

° R[rt] <- R[rs] op ZeroExt[imm16] ]

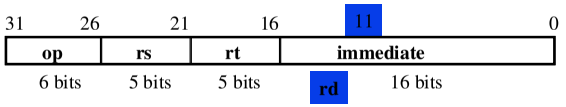


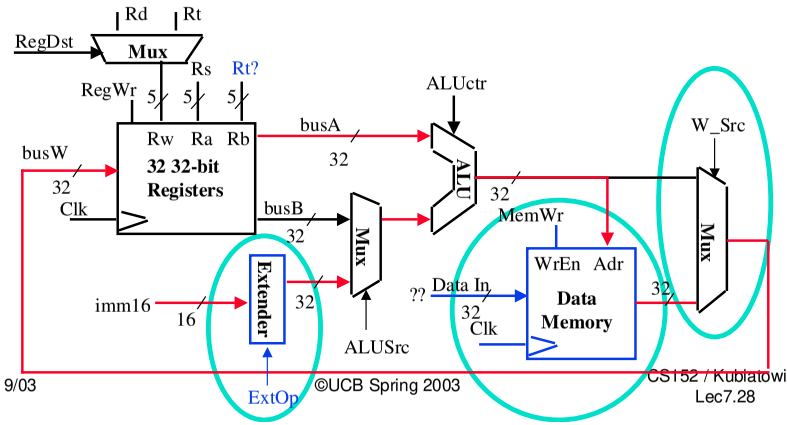






## 3d: Load Operations

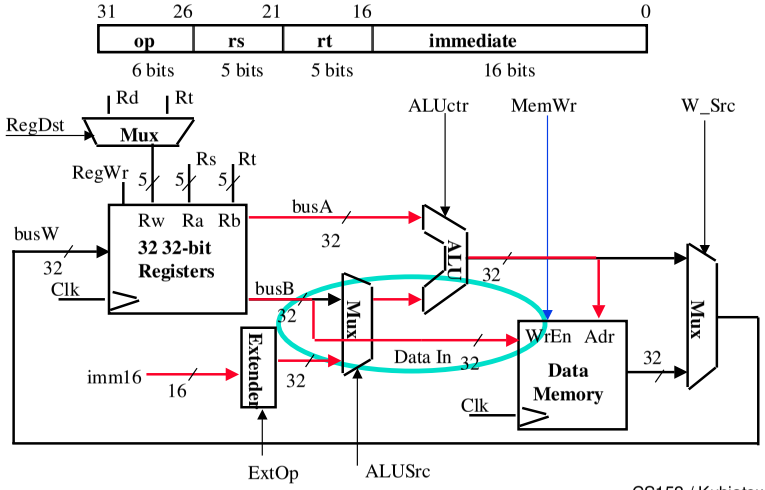
° R[rt] <- Mem[R[rs] + SignExt[imm16]]  Example: lw rt, rs, imm16



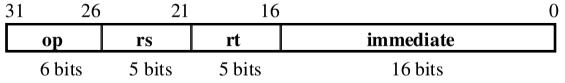
## 3e: Store Operations

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






## 3f: The Branch Instruction



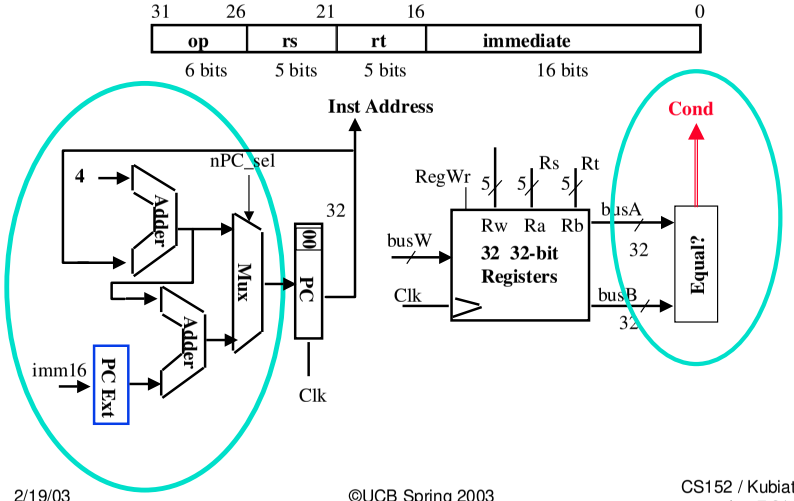

° beq  rs, rt, imm16

- mem[PC]  Fetch the instruction from memory
- Equal <- R[rs] == R[rt]  Calculate the branch condition
- if (Equal)  Calculate the next instruction's address
  - PC <- PC + 4 + ( SignExt(imm16) x 4 )
- else
  - PC <- PC + 4



## Datapath for Branch Operations

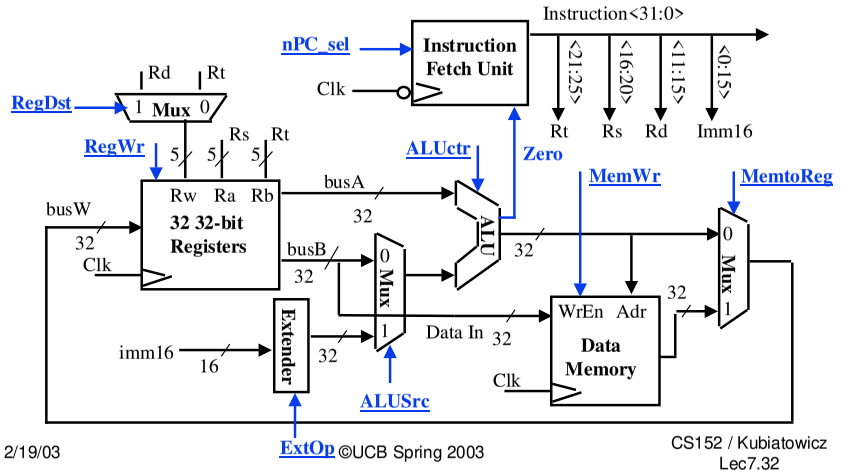
° beq  rs, rt, imm16  Datapath generates condition (equal)






## Putting it all together

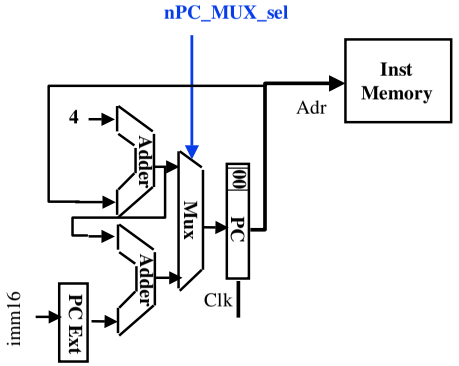
° Rs, Rt, Rd and Imed16 hardwired into datapath from Fetch Unit
° We have everything except control signals (underline)
  • Today's lecture will show you how to generate the control signals



## Recap: Meaning of the Control Signals

° nPC_MUX_sel: 0 ⇒ PC <- PC + 4
1 ⇒ PC <- PC + 4 + SignExt(Im16) || 00

° Later in lecture: higher-level connection between mux and branch cond






## Recap: Meaning of the Control Signals

° ExtOp: "zero", "sign"

° ALUsrc: 0 ⇒ regB; 1 ⇒ immed

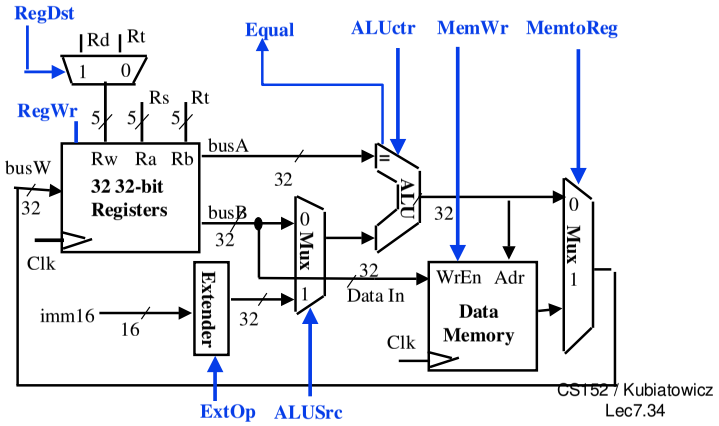
° ALUctr: "add", "sub", "or"

° MemWr: 1 ⇒ write memory

° MemtoReg: 0 ⇒ ALU; 1 ⇒ Mem

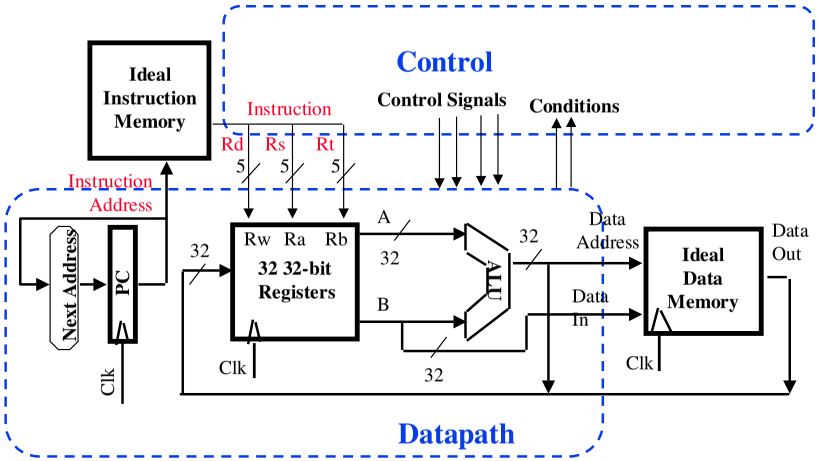
° RegDst: 0 ⇒ "rt"; 1 ⇒ "rd"

° RegWr: 1 ⇒ write register



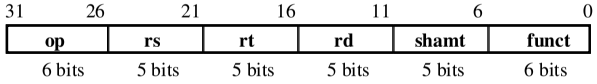



## Step 4: An Abstract View of the Implementation






## RTL: The Add Instruction

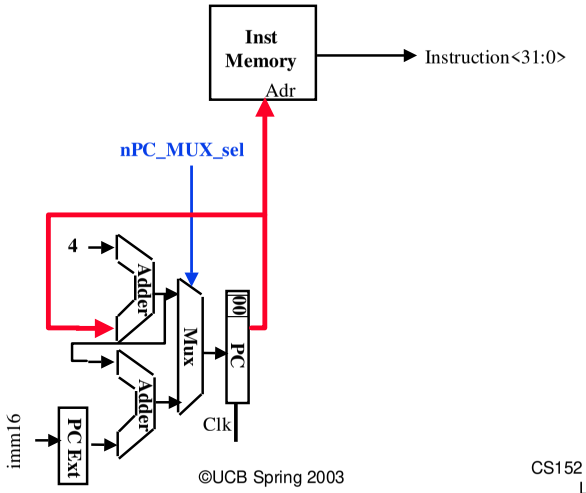



° add rd, rs, rt

- 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

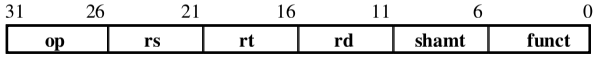
## Instruction Fetch Unit at the Beginning of Add

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

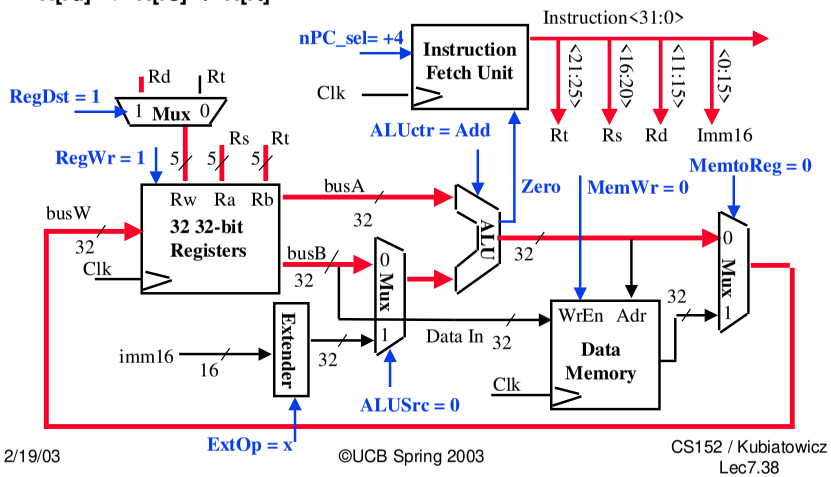





## The Single Cycle Datapath during Add



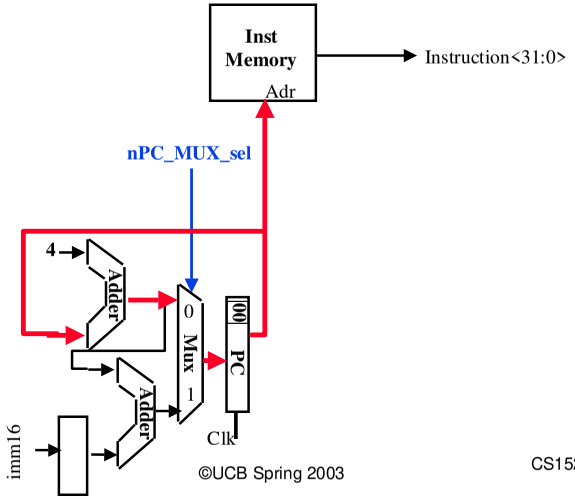

° R[rd] <- R[rs] + R[rt]





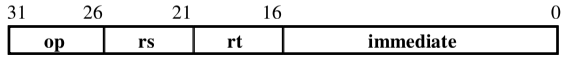

## Instruction Fetch Unit at the End of Add

° PC <- PC + 4
  • This is the same for all instructions except: Branch and Jump

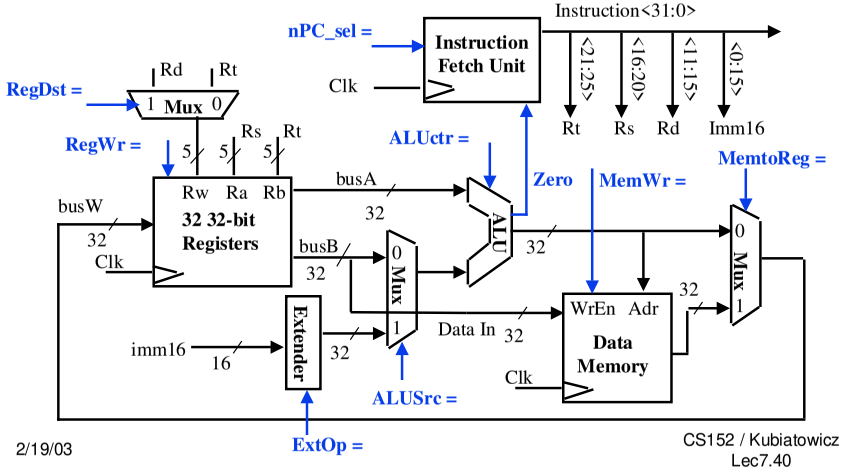




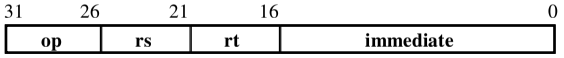

## The Single Cycle Datapath during Or Immediate



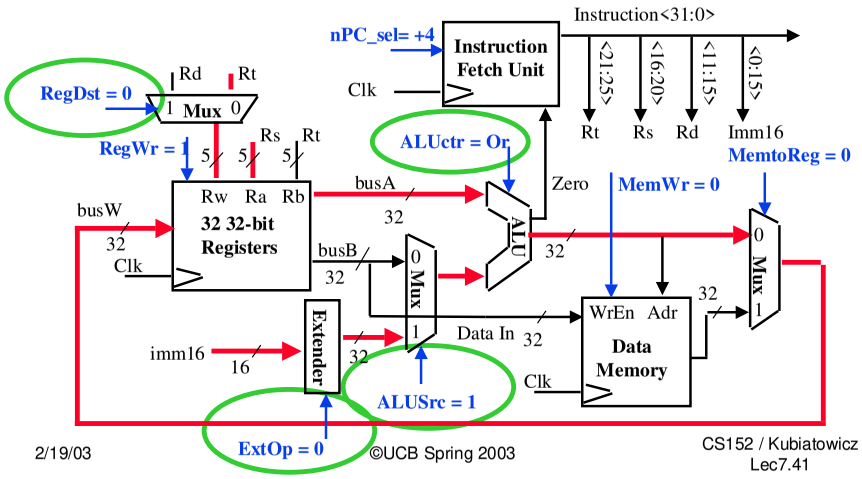

° R[rt] <- R[rs] or ZeroExt[Imm16]



## The Single Cycle Datapath during Or Immediate

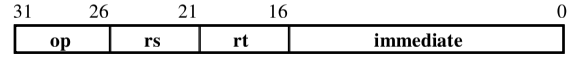



° R[rt] <- R[rs] or ZeroExt[Imm16]

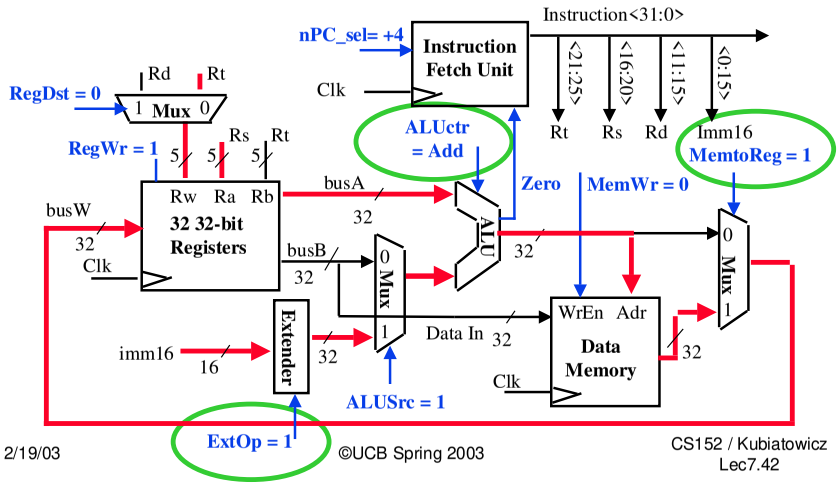





## The Single Cycle Datapath during Load

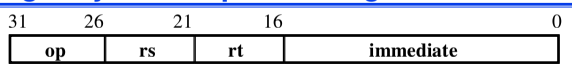



° R[rt] <- Data Memory {R[rs] + SignExt[imm16]}

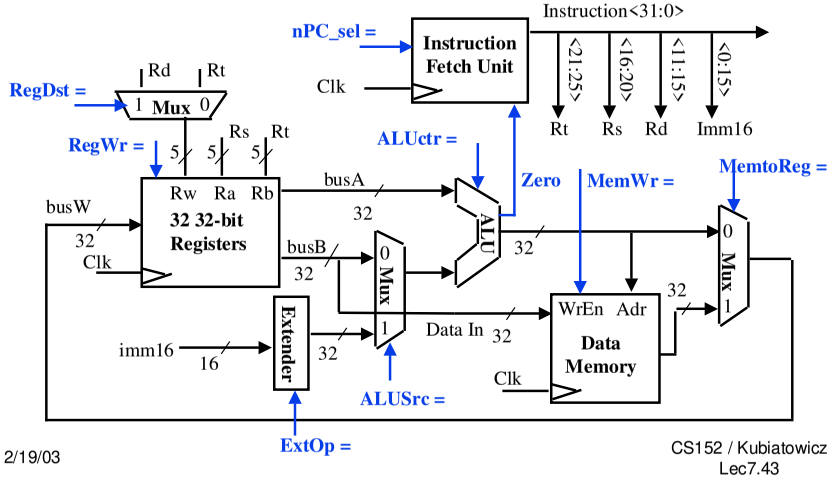





## The Single Cycle Datapath during Store

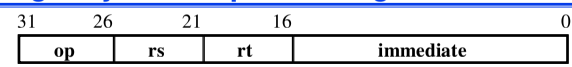



° Data Memory {R[rs] + SignExt[imm16]} <- R[rt]

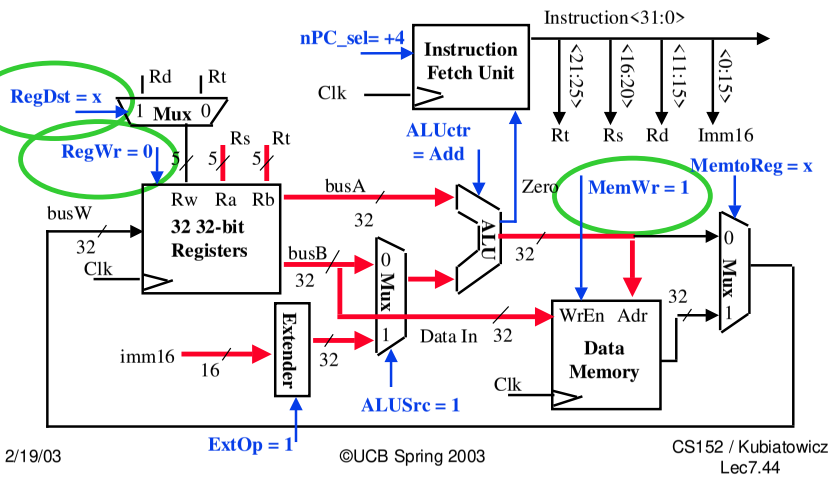




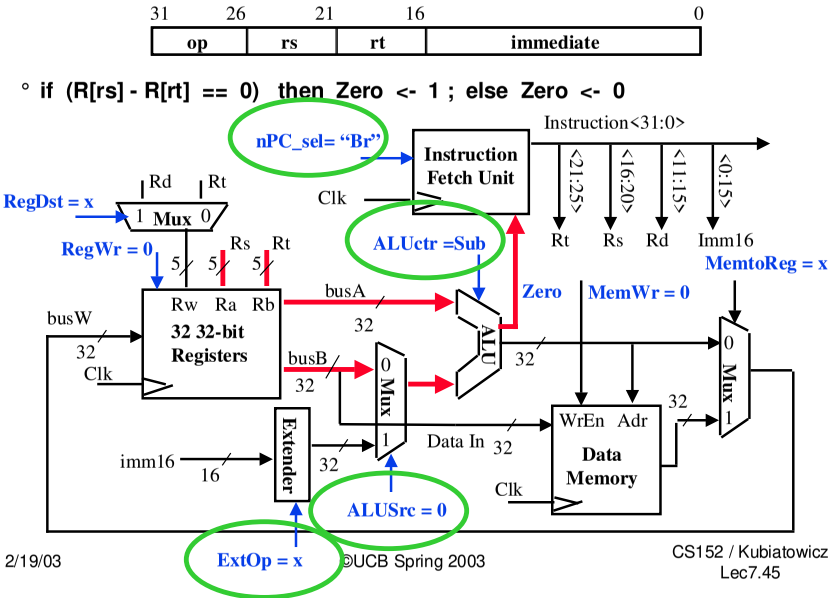

## The Single Cycle Datapath during Store




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

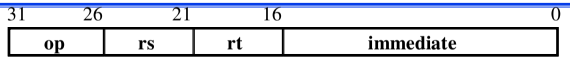


## The Single Cycle Datapath during Branch

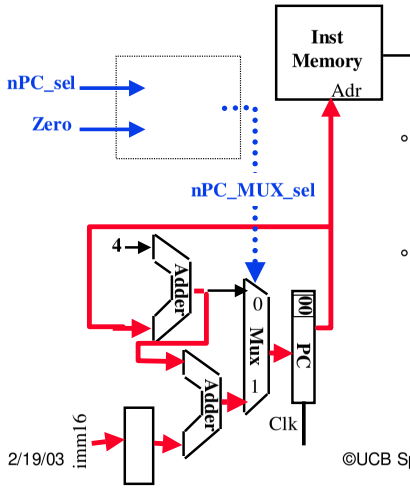





## Instruction Fetch Unit at the End of Branch




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




° What is encoding of nPC_sel?
- Direct MUX select?
- Branch / not branch

° Let's choose second option

| nPC_sel | zero? | MUX |
|---|---|---|
| 0 | x | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



## A Summary of Control Signals: The Decode Process

**This is the process of "decoding" instructions!**

| 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" | |

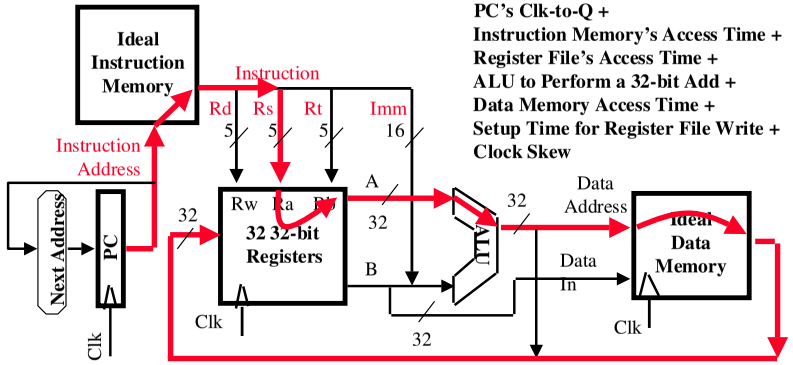


## Step 5: Assemble Control

° This is a hardware implementation issue
- We will complete design next lecture
- LAB 3 involves implementing this processor

## An Abstract View of the Critical Path

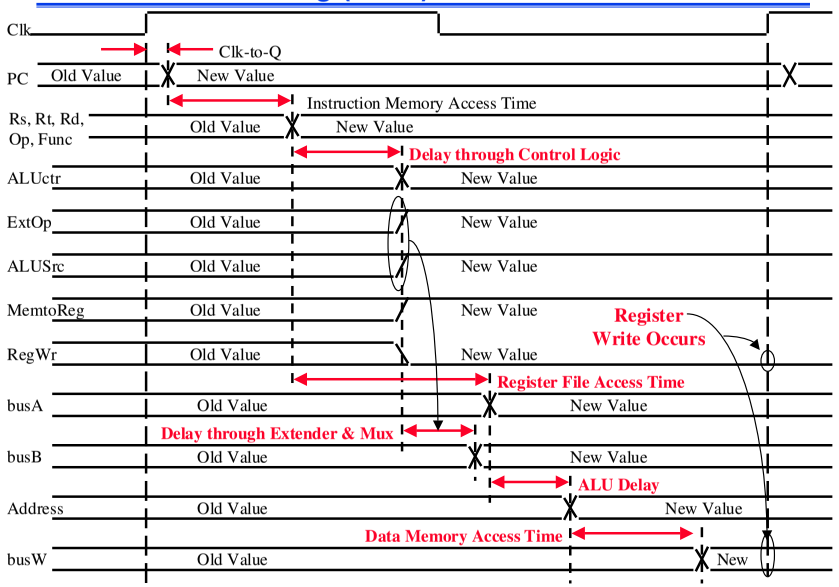
° Register file and ideal memory:
  • The CLK input is a factor ONLY during write operation
  • During read operation, behave as combinational logic:
    - Address valid => Output valid after "access time."




**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**



## Worst Case Timing (Load)




# Next Time:
# Multi-cycle Implementation!



## Summary

° 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

° Next time: implementing control