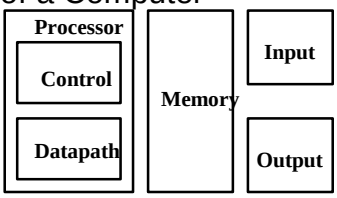
Ch 5: Designing a Single Cycle Datapath

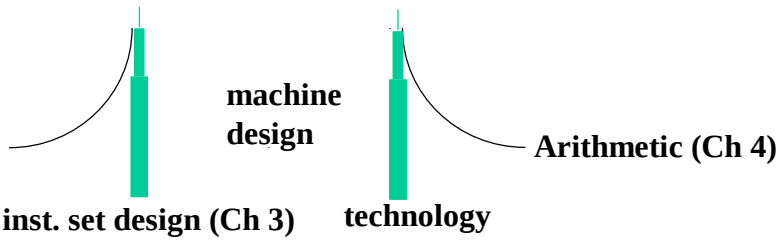
Computer Systems Architecture
CS 365

The Big Picture: Where are We Now?

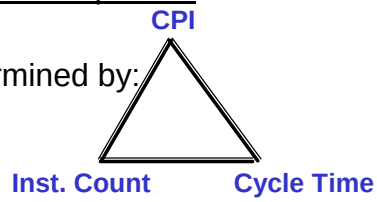
- The Five Classic Components of a Computer



- Today's Topic: Design a Single Cycle Processor



The Big Picture: The Performance Perspective



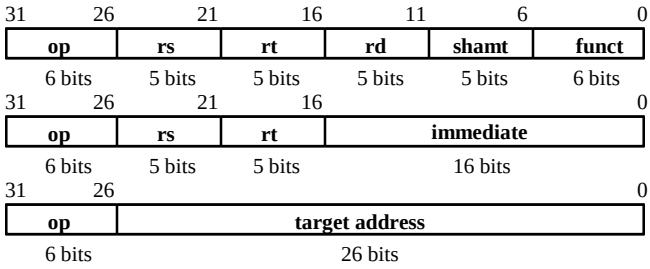
- Performance of a machine is determined by:
  - Instruction count
  - Clock cycle time
  - Clock cycles per instruction
- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time

How to Design a Processor: step-by-step

1. Analyze instruction set => datapath requirements
   - 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. 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

The MIPS Instruction Formats

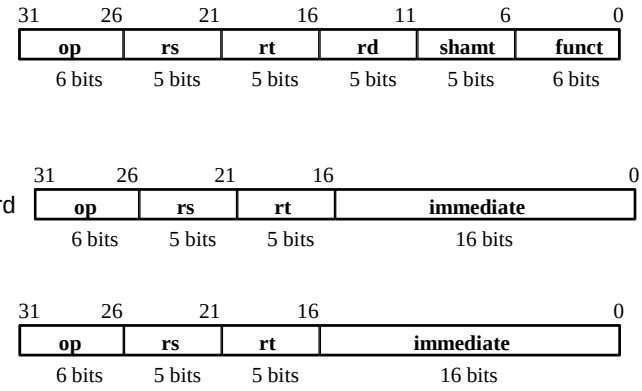
- All MIPS instructions are 32 bits long. The three instruction formats:
  - R-type
  - I-type
  - J-type



- 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

Step 1a: The MIPS-lite Subset

- ADD, SUB, AND, OR
  - add rd, rs, rt
  - sub rd, rs, rt
  - and rd, rs,rt
  - or rd,rs,rt
- 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
- First step is to fetch the instruction from memory

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

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

inst	Register Transfers	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
OR	R[rt] <– R[rs] | R[rt];	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 + sign_ext(Imm16)] || 00
	else	PC <– PC + 4

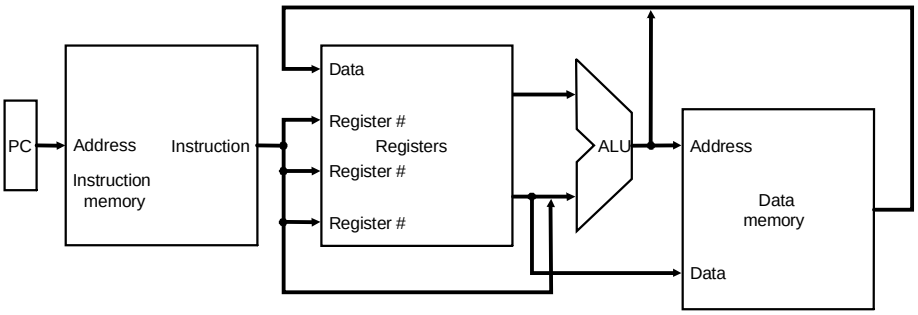
Step 1: Requirements of the Instruction Set

- 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

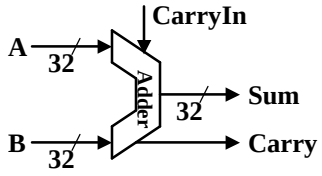
Abstract/Simplified View of Datapath



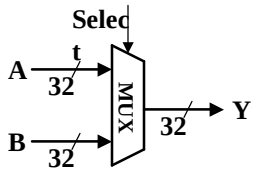
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

Combinational Logic Elements (Basic Building Blocks)

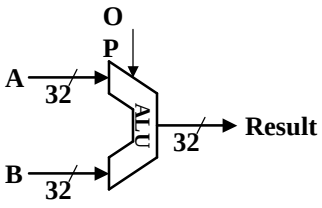
- Adder



- MUX

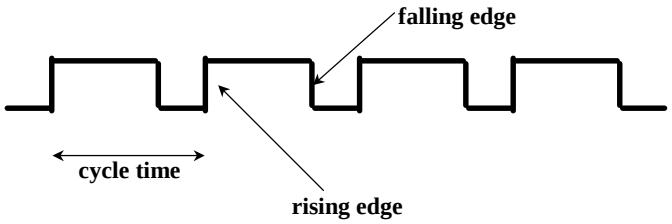


- ALU



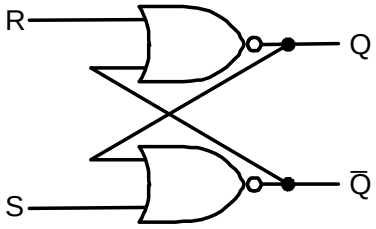
State Elements: Review

- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?



An unclocked state element

- The set-reset latch
  - output depends on present inputs and also on past inputs



Latches and Flip-flops

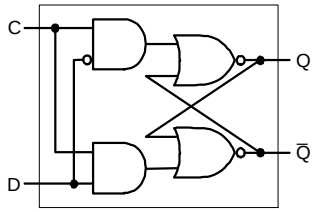
- Output is equal to the stored value inside the element
  (don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock
- Latches: whenever the inputs change, and the clock is asserted
- Flip-flop: state changes only on a clock edge
  (edge-triggered methodology)

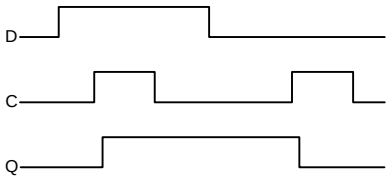
"logically true",
— could mean electrically low

A clocking methodology defines when signals can be read and written
— wouldn't want to read a signal at the same time it was being written

D-latch

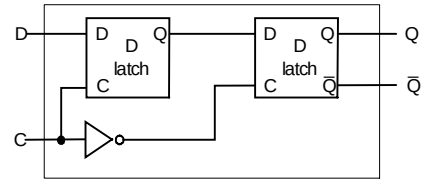
- Two inputs:
  - the data value to be stored (D)
  - the clock signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and its complement

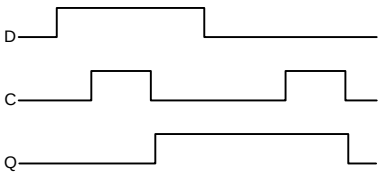




D flip-flop

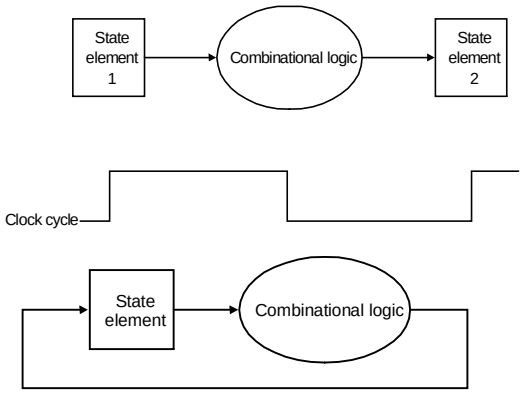
- Output changes only on the clock edge





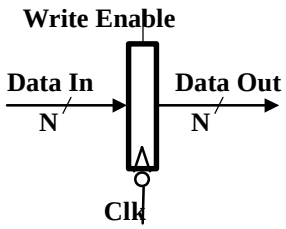
Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements



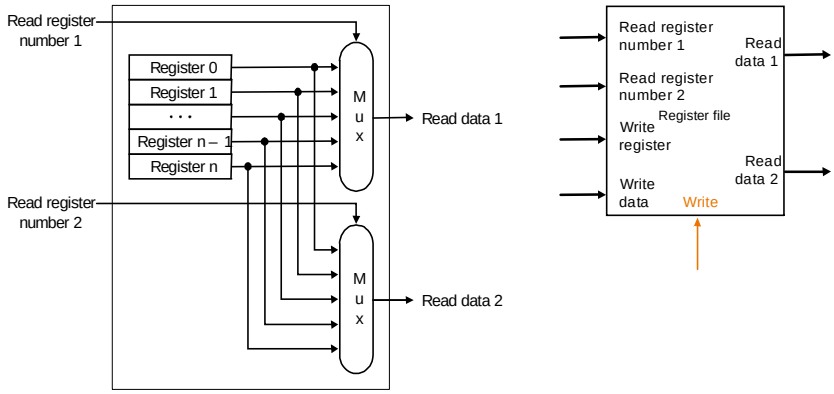
Storage Element: Register (Basic Building Block)

- 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



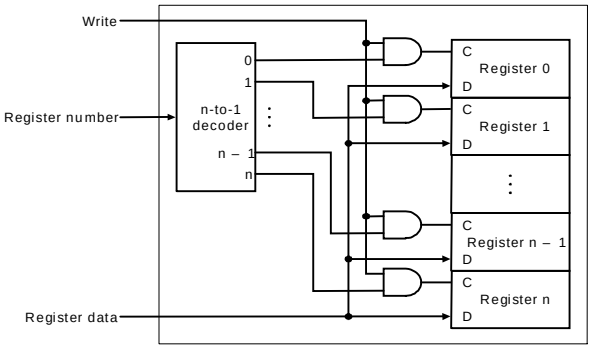
Register File

- Built using D flip-flops

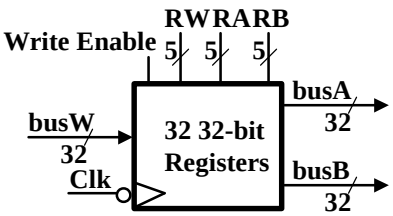


Register File

- Note: we still use the clock to determine when to write

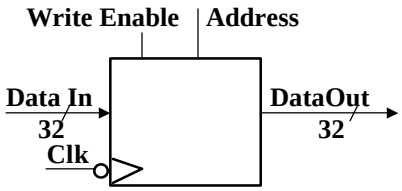


Storage Element: Register File



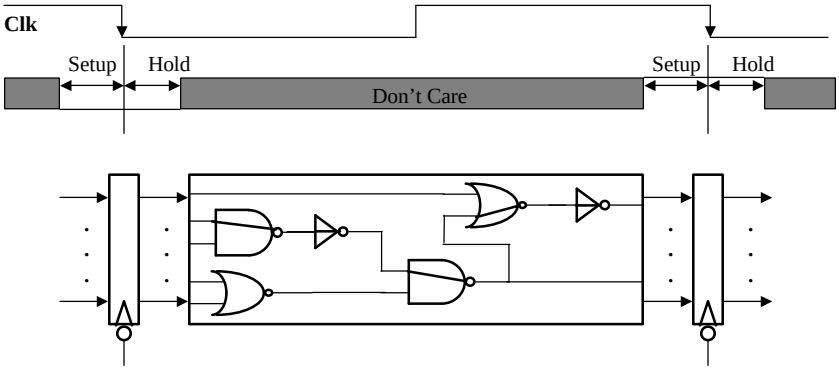
- Register File consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to 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."

Clocking Methodology



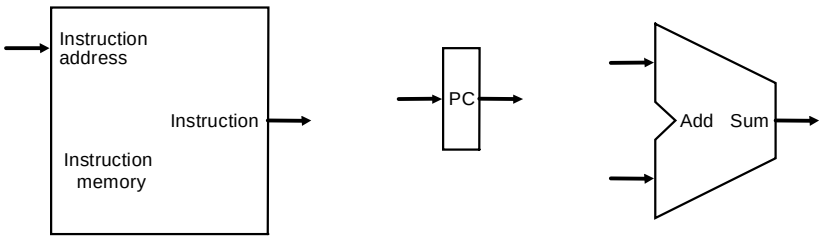
- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- (CLK-to-Q + Shortest Delay Path - Clock Skew) > Hold Time

Step 3

- Register Transfer Requirements
  –> Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation

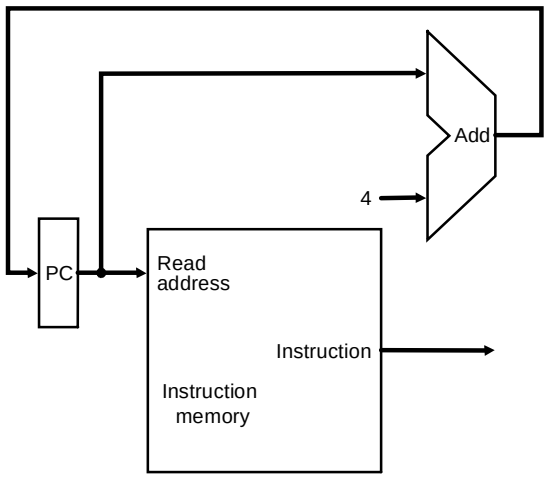
3a: Overview of the Instruction Fetch Unit

- The common RTL operations
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- “something else”
    - We don’t know if instruction is a Branch/Jump or one of the other instructions until we have fetched and interpreted the instruction from memory. So all instructions initially increment the PC



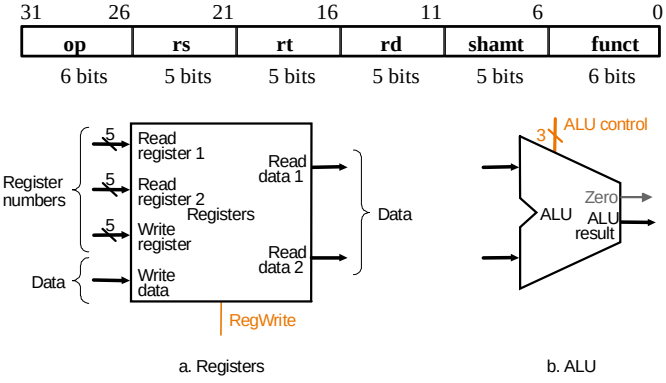
a. Instruction memory b. Program counter c. Adder

Datapath for Instruction Fetch

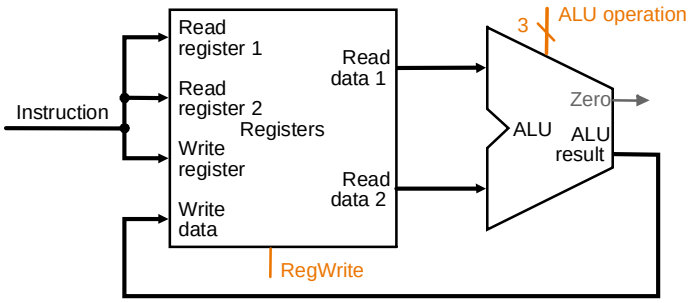


3b: R-format instructions: add, sub, and, or, slt

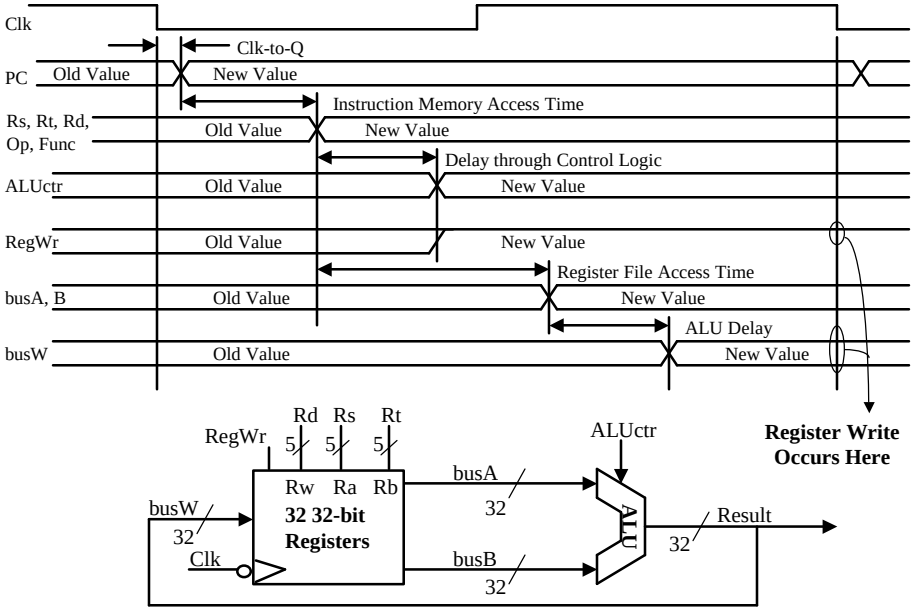
- R[rd] <- R[rs] op R[rt] Example: add rd, rs, rt
  - Read register 1, Read register 2, and Write register come from instruction's rs, rt, and rd fields
  - ALU control and RegWrite: control logic after decoding the instruction



Datapath for R-format instructions

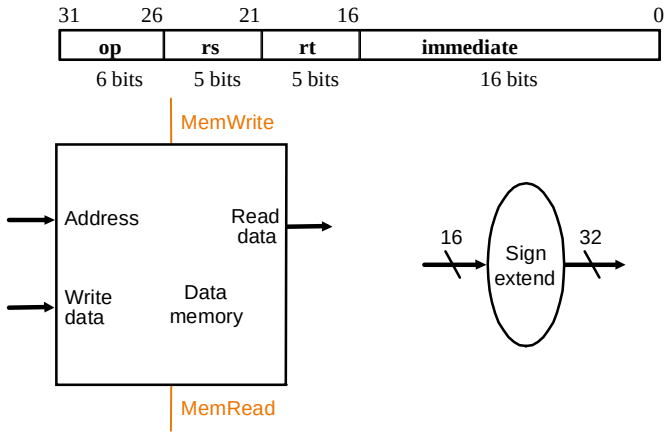


Register-Register Timing



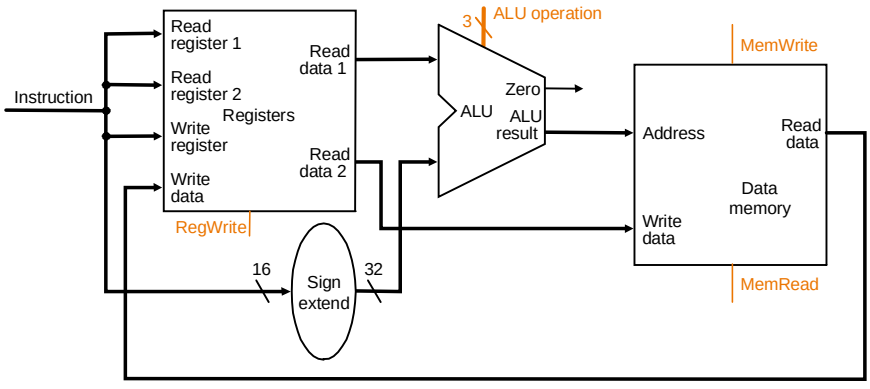
3d: Load & Store Operations

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

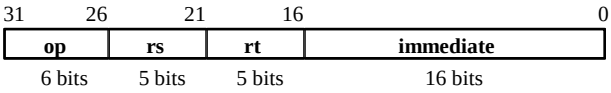


a. Data memory unit b. Sign-extension unit

Datapath for lw & sw



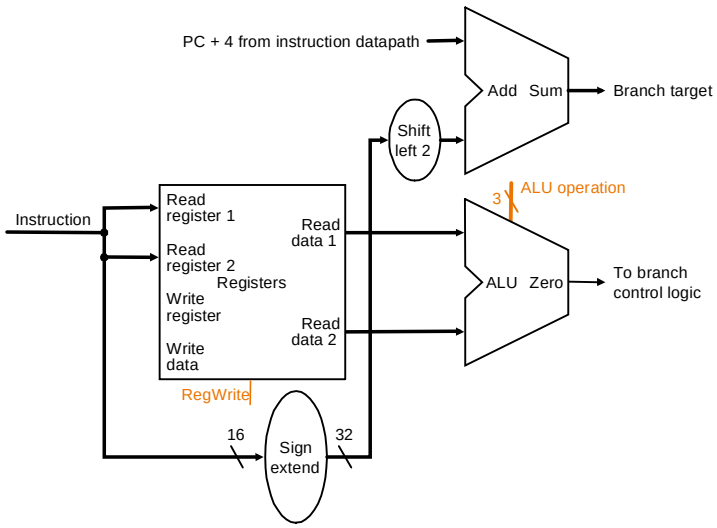
3f: The Branch Instruction



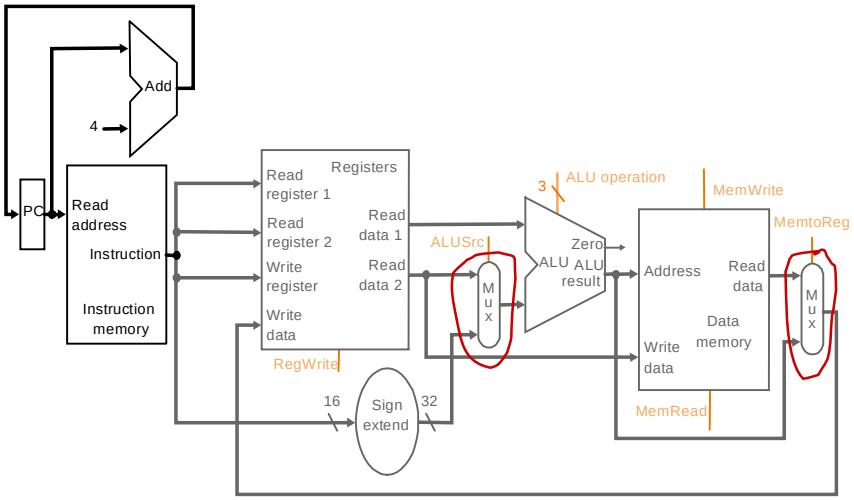
- beq rs, rt, imm16
  - mem[PC] Fetch the instruction from memory
  - Equal <- R[rs] == R[rt] Calculate the branch condition

    if (COND eq 0) Calculate the next instruction's address
    PC <- PC + 4 + (SignExt(imm16) x 4)
    else
    PC <- PC + 4

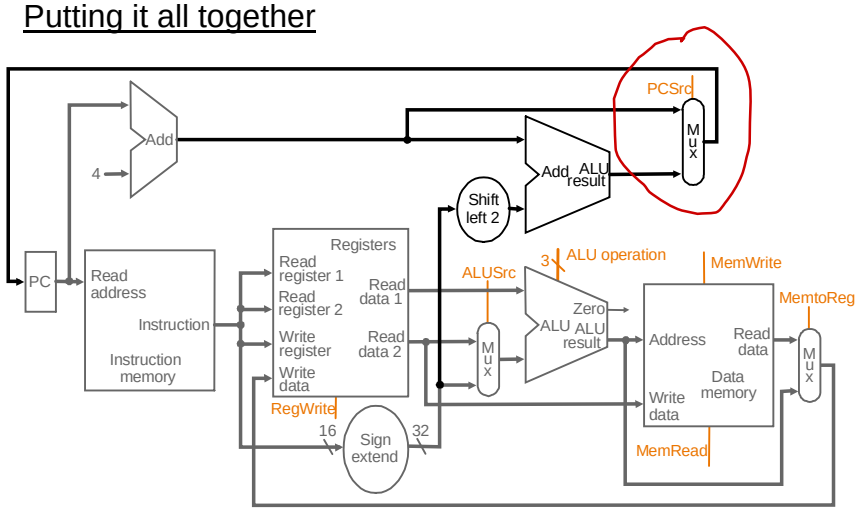
Datapath for branch instruction



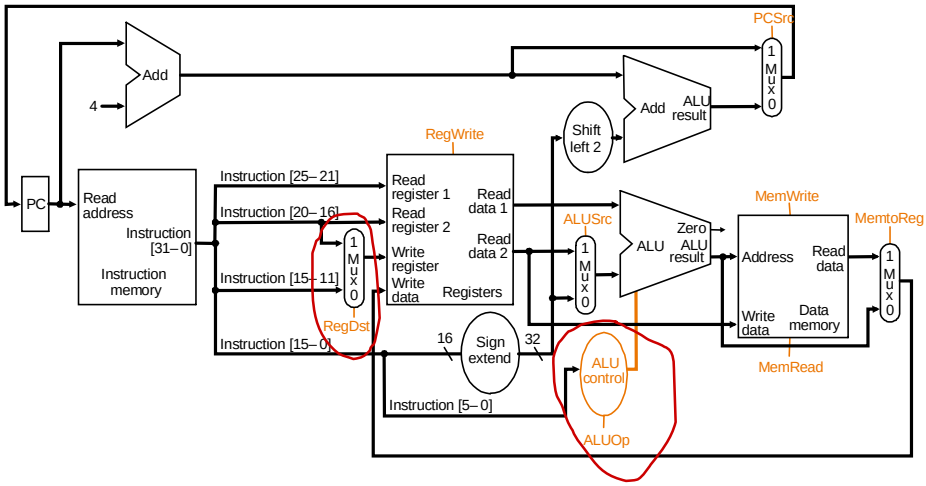
Using multiplexors to stitch together the datapath for memory access and R-format instructions



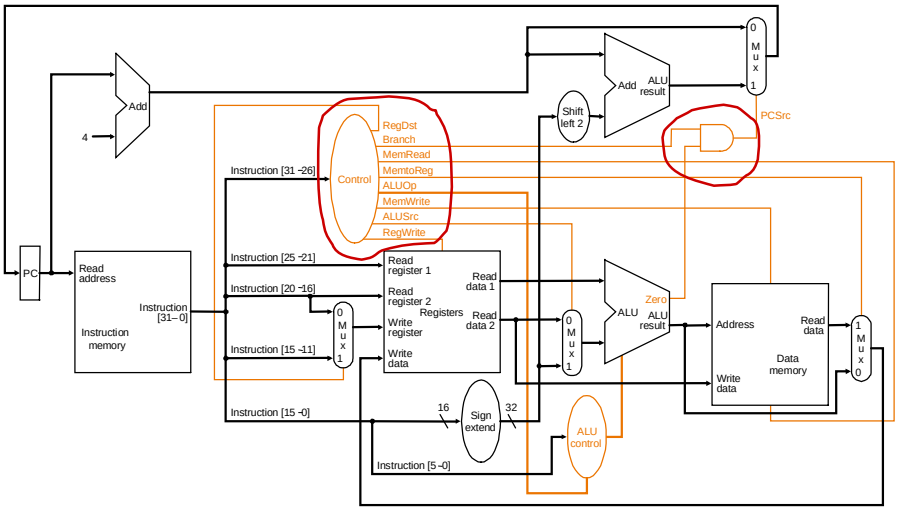
Putting it all together



Putting it all together cont'd



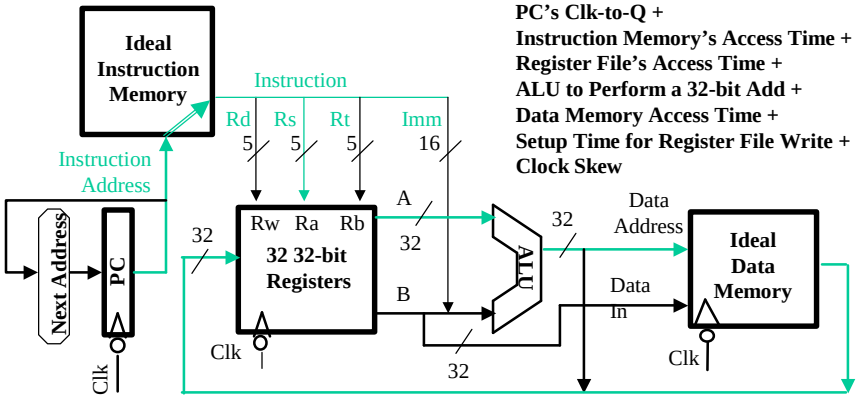
Adding the control unit



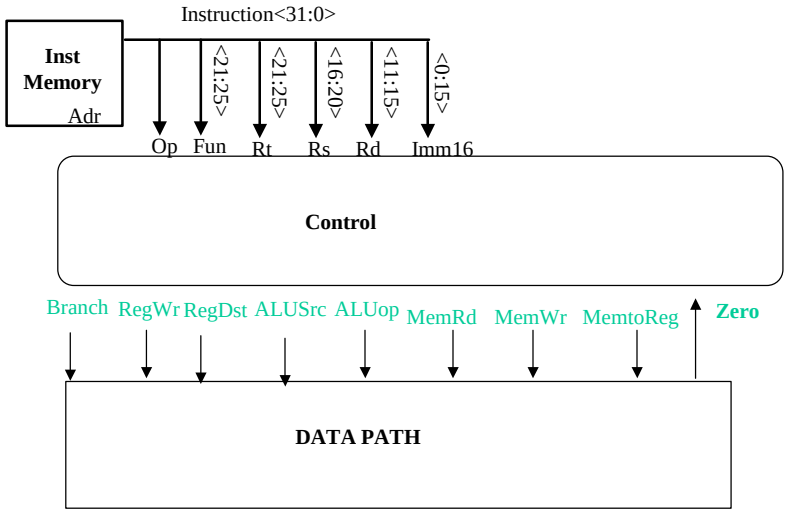
An Abstract View of the Critical Path

- 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



Step 4: Given Datapath: RTL -> Control



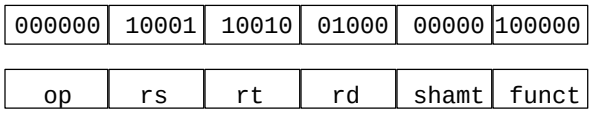
Control

- Selecting the operations to perform (ALU, read/write, etc.)

  Design the ALU Control Unit
- Controlling the flow of data (multiplexor inputs)

  Design the Main Control Unit
- Information comes from the 32 bits of the instruction
- Example:

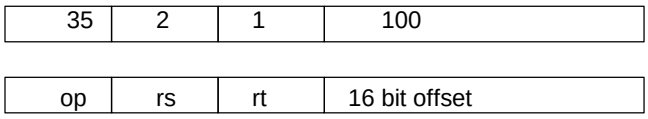
add $8, $17, $18 Instruction Format:



- ALU's operation based on instruction type and function code

ALU Control

- e.g., what should the ALU do with this instruction
- Example: lw $1, 100($2)



- ALU control input

```
000    AND
001    OR
010    add
110    subtract
111    set-on-less-than
```

- Why is the code for subtract 110 and not 011?)

  (Recall design of ALU from Chapter 4. Bnegate input for adder set to 1 for subtraction

ALU Control Design

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	Load word	xxxxxx	Add	010
SW	00	Store word	xxxxxx	Add	010
BEQ	01	Branch eq	xxxxxx	Subtract	110
R-type	10	Add	100000	Add	010
R-type	10	Subtract	100010	Subtract	110
R-type	10	AND	100100	And	000
R-type	10	OR	1000101	Or	001
R-type	10	Set on less than	101010	Set on less than	111

Control

- Must describe hardware to compute 3-bit ALU control input
  - given instruction type
    00 = lw, sw
    01 = beq
    10 = arithmetic
    ALUOp computed from instruction type
  - function code for arithmetic
- Describe it using a truth table (can turn into gates):

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Design the main control unit

- Seven control signals
  - RegDst
  - RegWrite
  - ALUSrc
  - PCSrc
  - MemRead
  - MemWrite
  - MemtoReg

Control Signals

1. RegDst = 0 => Register destination number for the Write register comes from the rt field (bits 20-16)

   RegDst = 1 => Register destination number for the Write register comes from the rd field (bits 15-11)
2. RegWrite = 1 => The register on the Write register input is written with the data on the Write data input (at the next clock edge)
3. ALUSrc = 0 => The second ALU operand comes from Read data 2

   ALUSrc = 1 => The second ALU operand comes from the sign-extension unit
4. PCSrc = 0 => The PC is replaced with PC+4

   PCSrc = 1 => The PC is replaced with the branch target address
5. MemtoReg = 0 => The value fed to the register write data input comes from the ALU

   MemtoReg = 1 => The value fed to the register write data input comes from the data memory
6. MemRead = 1 => Read data memory
7. MemWrite = 1 => Write data memory

R-format instructions

RegDst = 1
RegWrite = 1
ALUSrc = 0
Branch = 0
MemtoReg = 0
MemRead = 0
MemWrite = 0
ALUOp = 10

Memory access instructions

Load word

RegDst = 0
RegWrite = 1
ALUSrc = 1
Branch = 0
MemtoReg = 1
MemRead = 1
MemWrite = 0
ALUOp = 00

0

Store Word

RegDst = X
RegWrite = 0
ALUSrc = 1
Branch = 0
MemtoReg = X
MemRead = 0
MemWrite = 1
ALUOp = 00

Branch Equal

RegDst = X

RegWrite = 0

ALUSrc = 0

Branch = 1

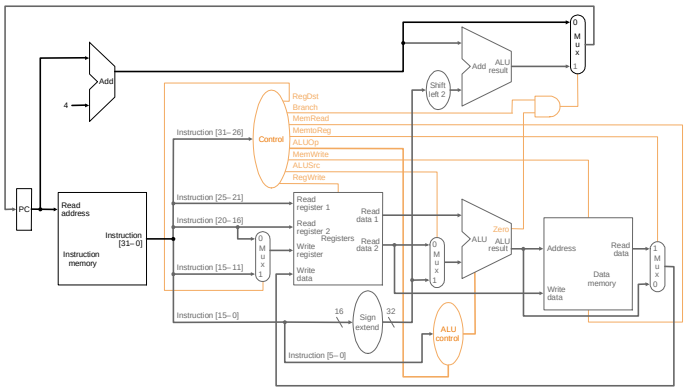
MemtoReg = X

MemRead = 0

MemWrite = 0

ALUOp = 01

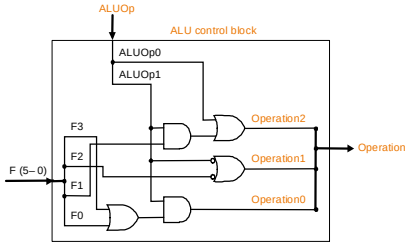
Control



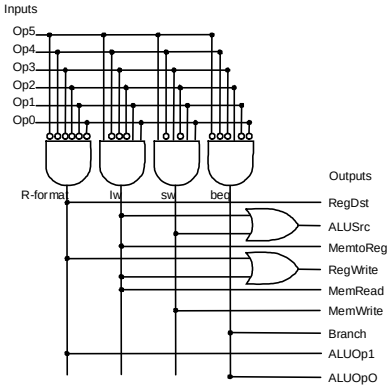
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Step 5: Implementing Control

- Simple combinational logic (truth tables)



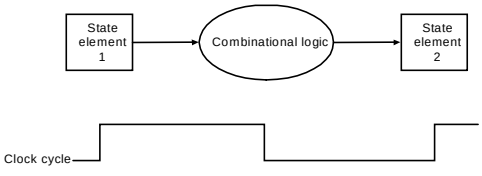
ALU Control Unit



Main Control Unit

Our Simple Control Structure

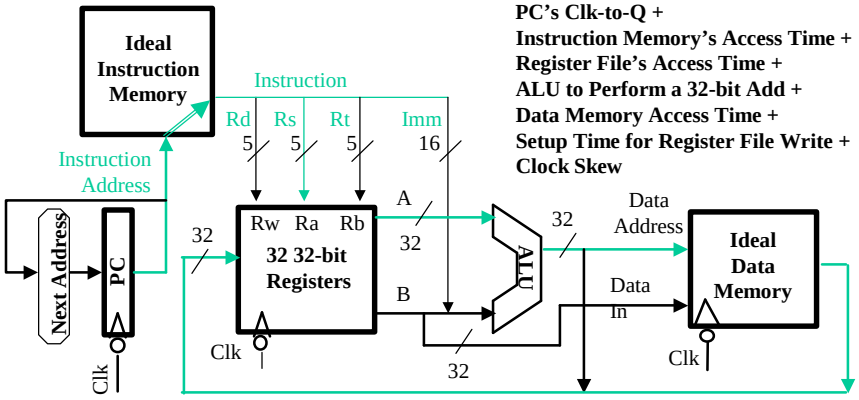
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



An Abstract View of the Critical Path

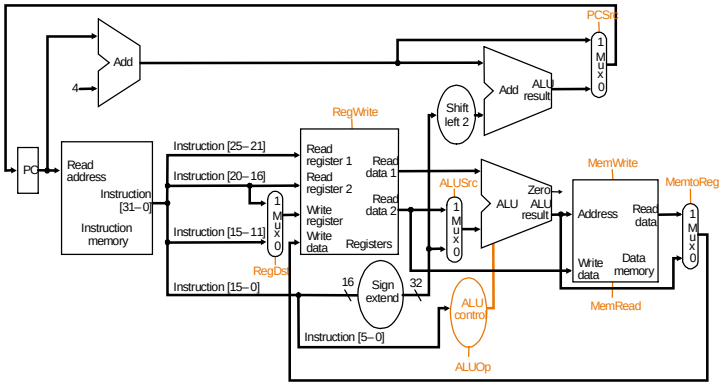
- Register file and ideal memory:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after "access time."

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

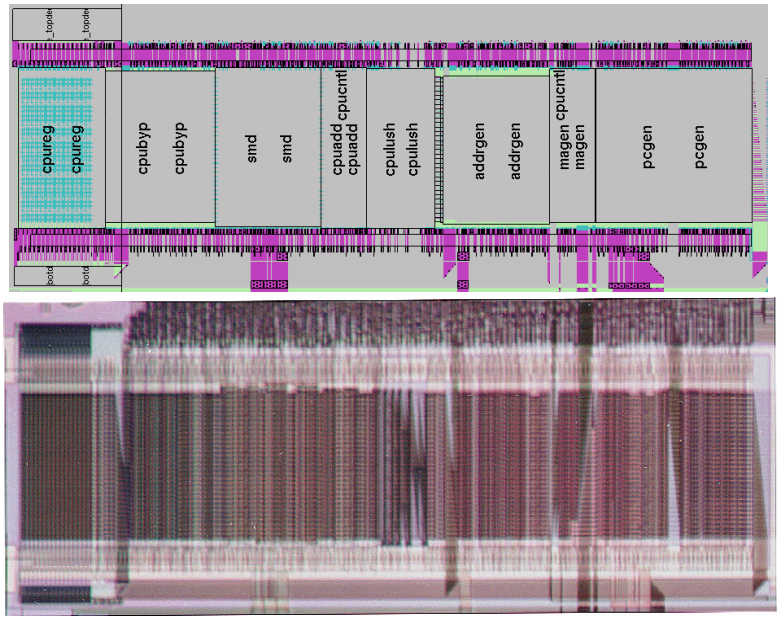


Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)



A Real MIPS Datapath (CNS T0)



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, Clock Cycle Time => long