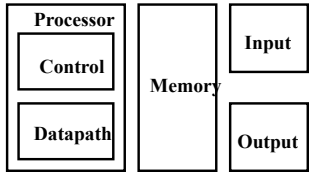
EEL-4713
Computer Architecture
Designing a Single Cycle Datapath

Outline

- Introduction
- The steps of designing a processor
- Datapath and timing for register-register operations
- Datapath for logical operations with immediates
- Datapath for load and store operations
- Datapath for branch and jump operations

Big Picture

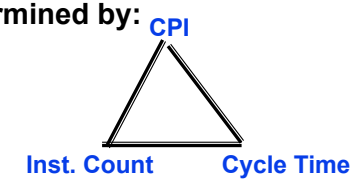
- The five classic components of a computer



- Today's topic: design of 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
    - CPI – will discuss later



- Processor design determines:
  - Clock cycle time
  - Clock cycles per instruction
- 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*
  • The datapath must include storage element for ISA registers
    - And possibly more
  • The datapath must support each register transfer

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

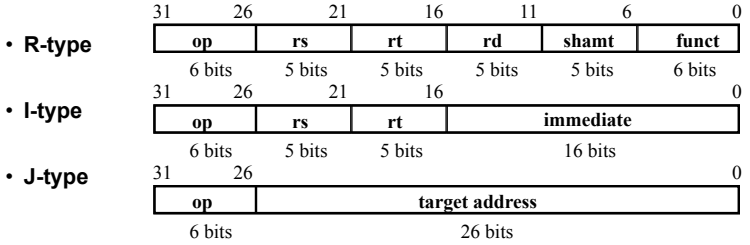
° 3. Assemble datapath meeting the requirements

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

° 5. Assemble the control logic

MIPS ISA: instruction formats

° All MIPS instructions are 32 bits long. There are 3 instruction formats:



° The different fields are:
  • op: operation of the instruction
  • rs, rt, rd: the source(s) 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

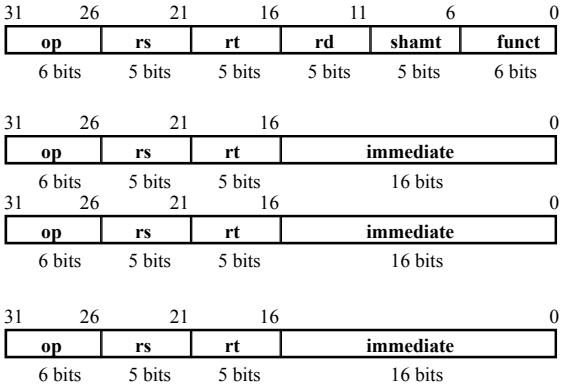
*Step 1a: The MIPS "lite" subset for today

° 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 + sign_ext(Imm16)]	
	else PC <– PC + 4	

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 + 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

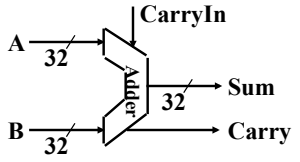
Step 2: Components of the Datapath

° Combinational Elements

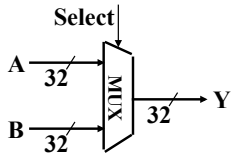
° Storage Elements
- Clocking methodology

Combinational Logic Elements (Basic Building Blocks)

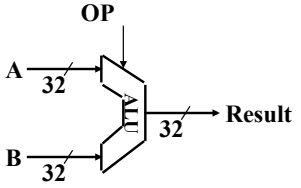
° Adder



° MUX

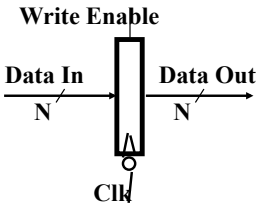


° ALU



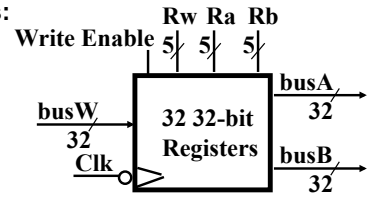
Storage Element: Register (Basic Building Block)

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



Storage Element: Register File

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

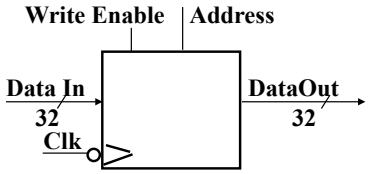


° Register is selected by:
- Ra (number) selects the register to put on busA (data)
- Rb (number) selects the register to put on busB (data)
- Rw (number) selects the register to be written via busW (data) when Write Enable is 1

° Clock input (CLK)
- The CLK input is a factor ONLY during write operation
- Read operations behave as a combinational logic block (i.e., reads are not clocked):
  - RA or RB valid => busA or busB valid after "access time."

Storage Element: Idealized Memory

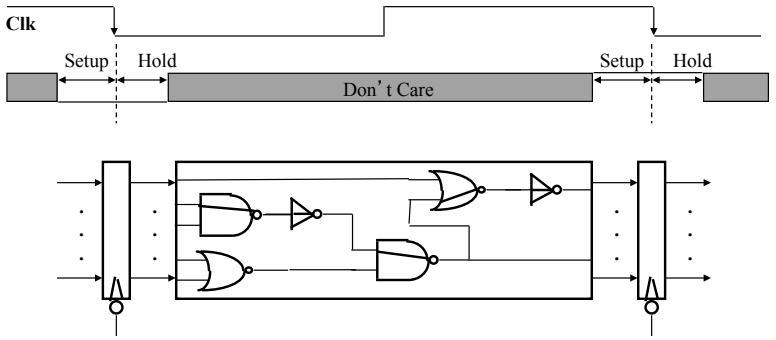
° Memory (idealized)
- One input bus: Data In
- One output bus: Data Out



° 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
- Read operations behave as a combinational logic block (i.e., reads are not clocked):
  - Address valid => Data Out valid after "access time."

Clocking Methodology



° All storage elements are clocked by the same clock edge

° Cycle Time = Hold + Longest Delay Path + Setup + Clock Skew

Step 3

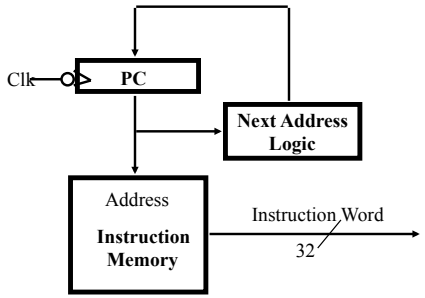
° Register Transfer Requirements –> Datapath Assembly

° Instruction Fetch

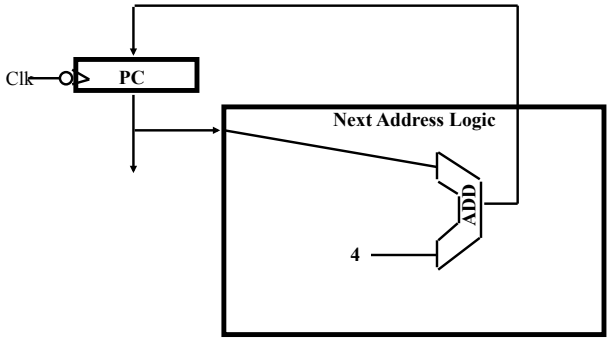
° Read Operands and Execute Operation

3a: Overview of the Instruction Fetch Unit

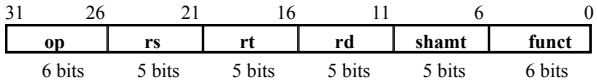
° The common RTL operations
- Fetch the Instruction: mem[PC]
- Update the program counter:
  - Sequential Code: PC <- PC + 4
  - Branch and Jump: PC <- "something else"



Next Address Logic – No Branching



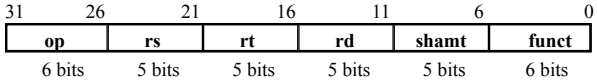
RTL: The ADD Instruction



° add rd, rs, rt

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

RTL: The Subtract Instruction



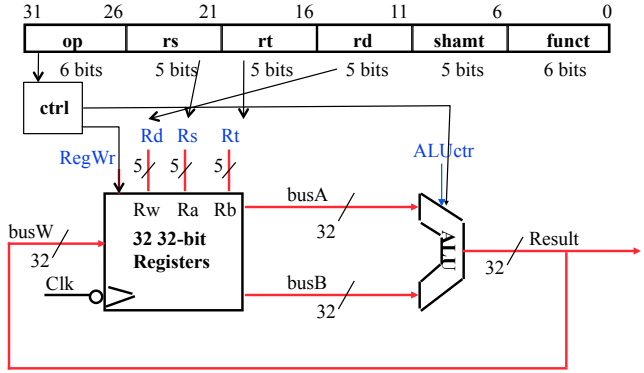
° sub rd, rs, rt

- op | rs | rt | rd | shamt | funct <- 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

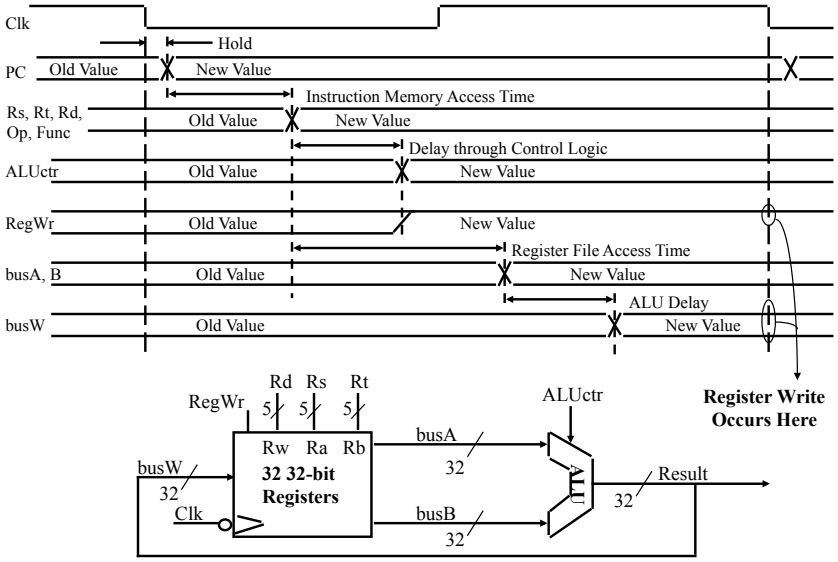
3b: Add & Subtract

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

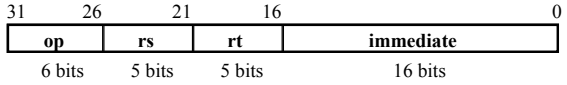
- Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
- ALUctr and RegWr: control logic after decoding the instruction



Register-Register Timing



RTL: The OR Immediate Instruction

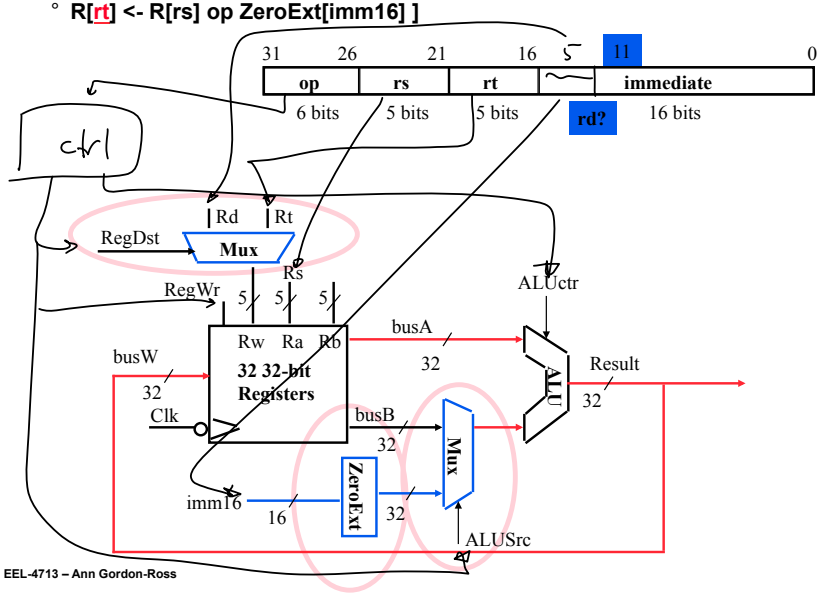


° ori rt, rs, imm16

- op | rs | rt | 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

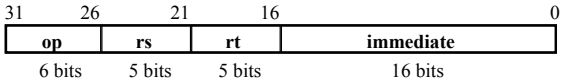
*3c: Logical Operations with Immediate

° R[rt] <- R[rs] op ZeroExt[imm16]]



RTL: The Load Instruction

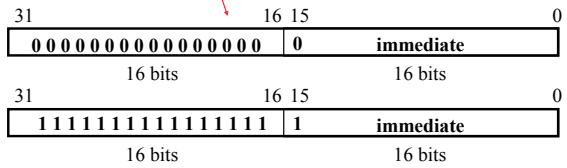
° lw rt, rs, imm16



- op | rs | rt | Imm16 <- mem[PC] Fetch the instruction from memory
- Addr <- R[rs] + SignExt(imm16) Calculate the memory address

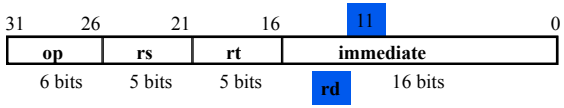
R[rt] <- Mem[Addr] Load the data into the register

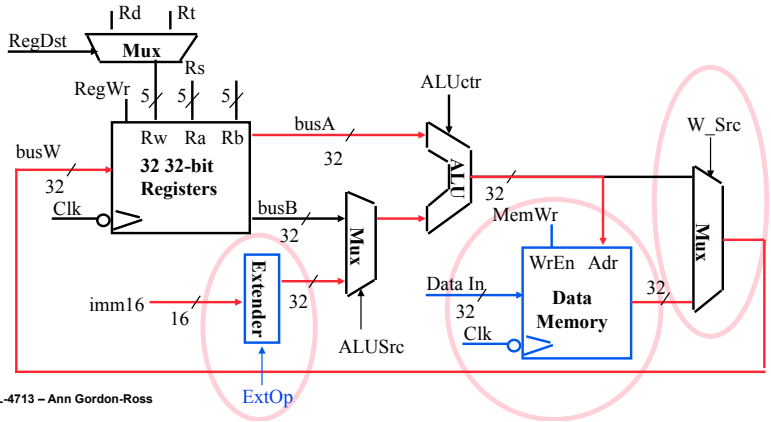
- PC <- PC + 4 Calculate the next instruction's address



3d: Load Operations

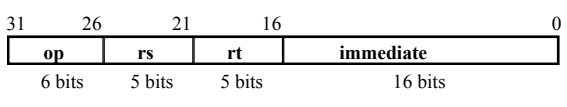
° R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: lw rt, rs, imm16

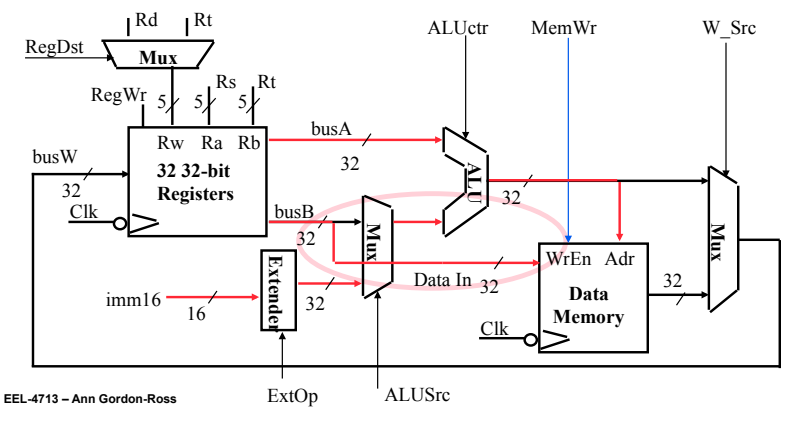




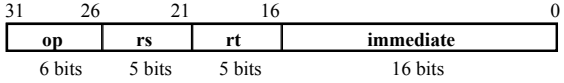
3e: Store Operations

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





3f: The Branch Instruction

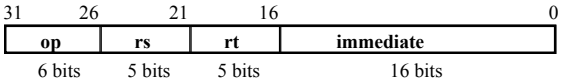


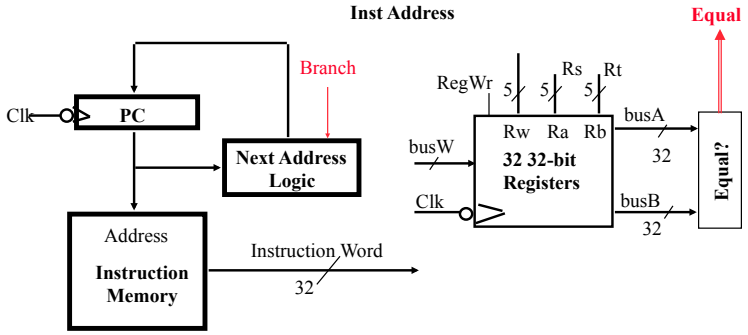
° beq rs, rt, imm16

- op | 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

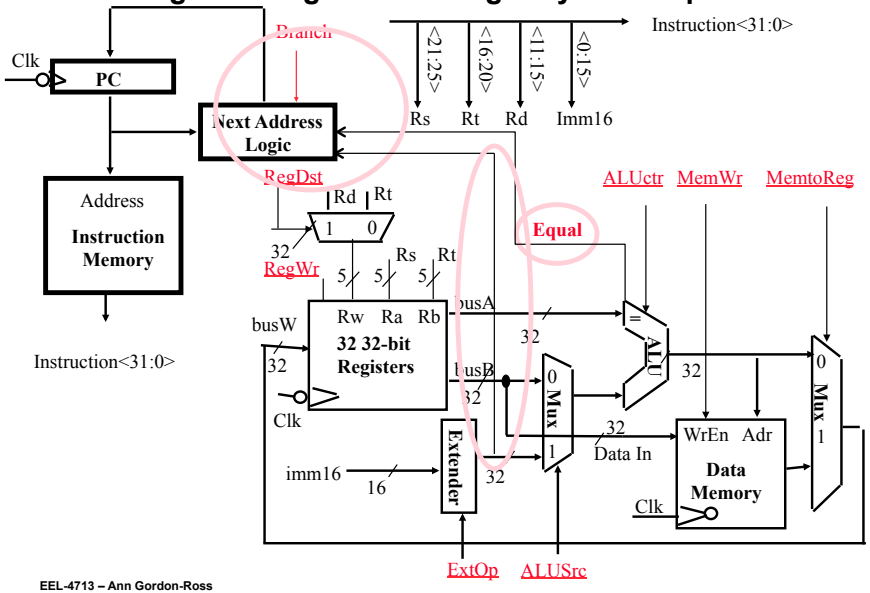
Datapath for Branch Operations

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

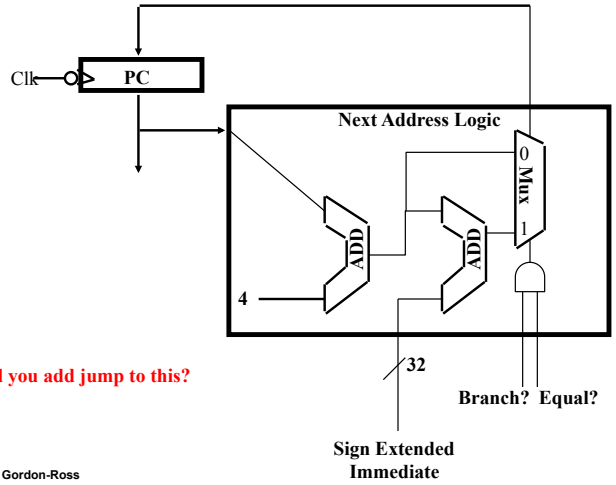




Putting it All Together: A Single Cycle Datapath



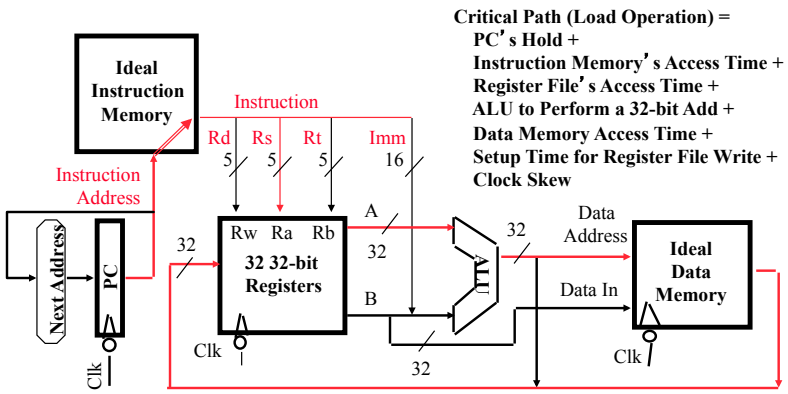
Next Address Logic – With Branching



How would you add jump to this?

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."



Binary arithmetic for the next address

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

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

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

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

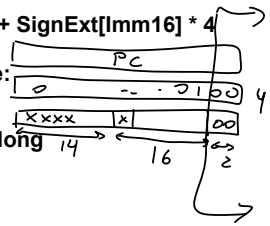
Binary arithmetic for the next address

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

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

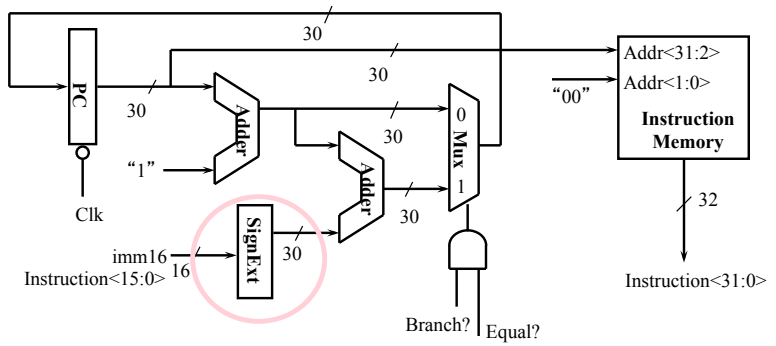
° In other words:
  • The 2 LSBs of the 32-bit PC are always zeros
  • There is no reason to have hardware to keep the 2 LSBs

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



Next Address Logic: Expensive and Fast Solution

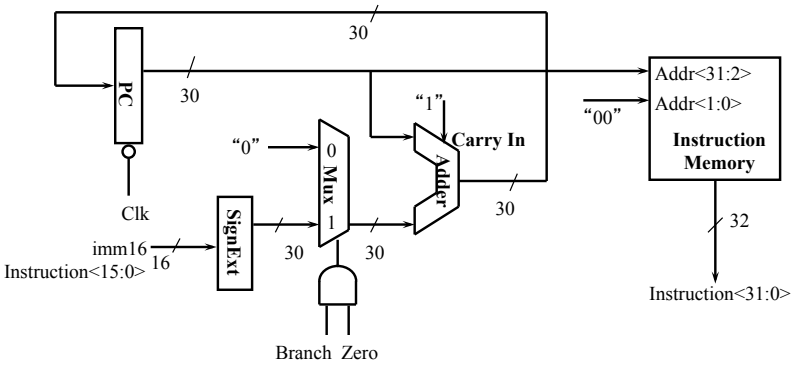
° Using a 30-bit PC:
  • Sequential operation: PC<31:2> = PC<31:2> + 1
  • Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  • In either case: Instruction Memory Address = PC<31:2> concat "00"



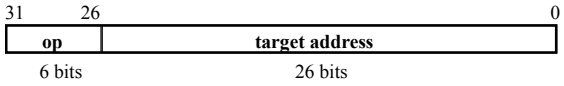
Next Address Logic: Cheap and Slow Solution

° 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.



RTL: The Jump Instruction

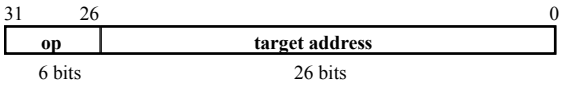


° j target

  • mem[PC] Fetch the instruction from memory

  • PC<31:2> <- PC<31:28> concat target<25:0>
    Calculate the next instruction's address

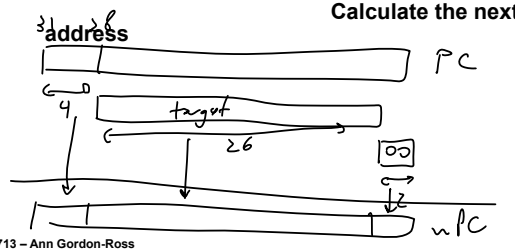
RTL: The Jump Instruction



° j target

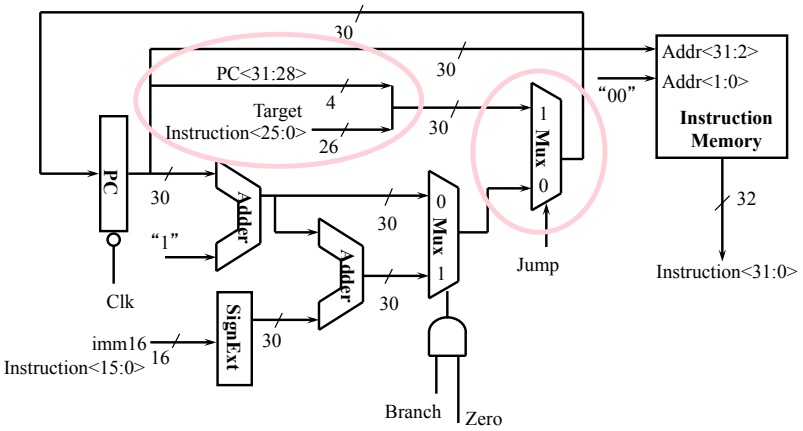
  • mem[PC] Fetch the instruction from memory

  • PC<31:2> <- PC<31:28> concat target<25:0>
    Calculate the next instruction's



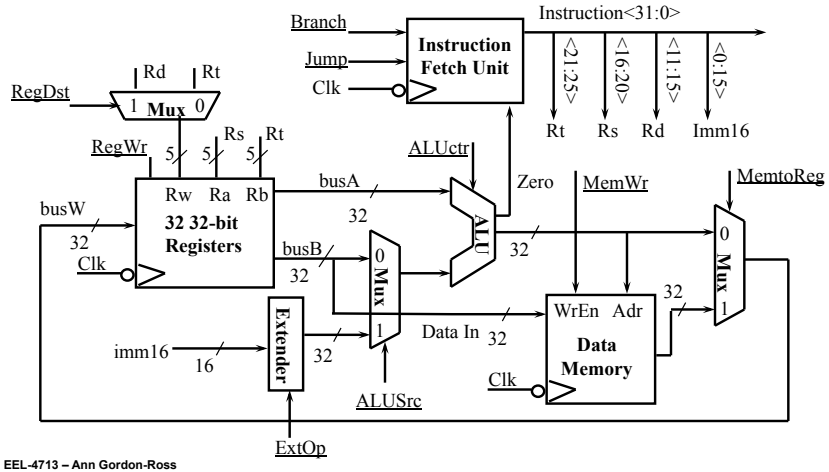
Instruction Fetch Unit

° j target
  • PC<31:2> <- PC<31:28> concat target<25:0>

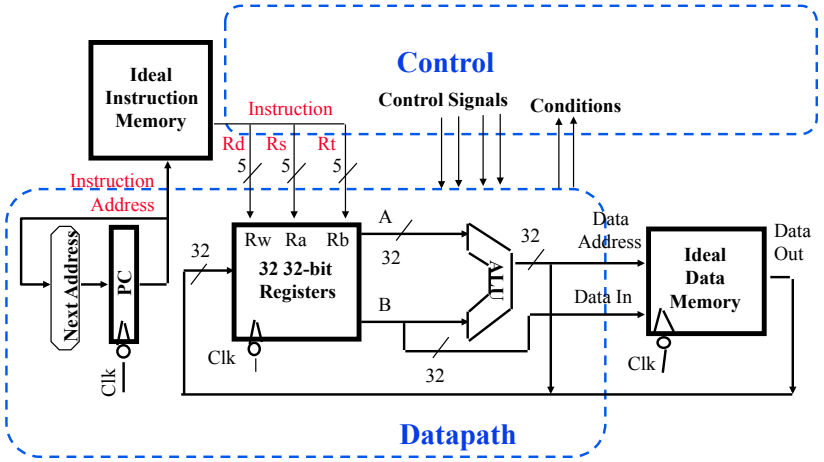


Putting it All Together: A Single Cycle Datapath

° We have everything except control signals (underline)



An Abstract View of the Implementation



° Logical vs. Physical Structure

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 (Steps 4 and 5)