

2009-07-22

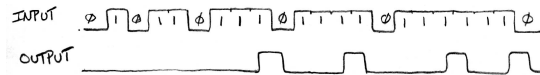


Jeremy Huddleston

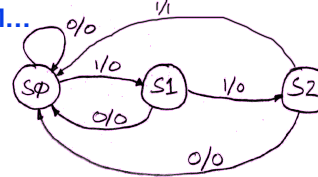


**Finite State Machine Example: 3 ones...**

FSM to detect the occurrence of 3 consecutive 1's in the input.



Draw the FSM...

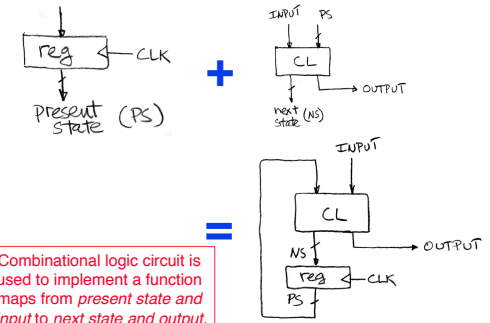


Assume state transitions are controlled by the clock: on each clock cycle the machine checks the inputs and moves to a new state and produces a new output...



**Hardware Implementation of FSM**

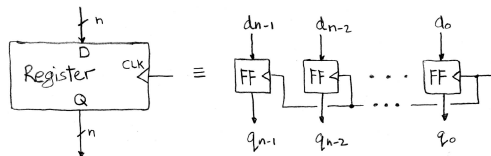
... Therefore a register is needed to hold the a representation of which state the machine is in. Use a unique bit pattern for each state.



Combinational logic circuit is used to implement a function maps from present state and input to next state and output.



**Register Details...What's inside?**

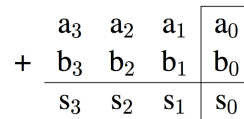


- n instances of a "Flip-Flop"
- Flip-flop name because the output flips and flops between and 0,1
- D is "data", Q is "output"

Also called "d-type Flip-Flop"



**Adder/Subtractor – One-bit adder LSB...**



a <sub>0</sub>	b <sub>0</sub>	s <sub>0</sub>	c <sub>1</sub>
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

s<sub>0</sub> =  
 c<sub>1</sub> =



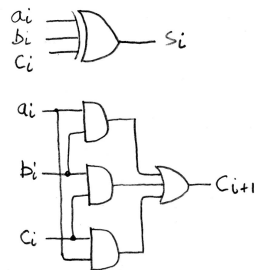
**Adder/Subtractor – One-bit adder (1/2)...**

a <sub>i</sub>	b <sub>i</sub>	c <sub>i</sub>	s <sub>i</sub>	c <sub>i+1</sub>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

s<sub>i</sub> =  
 c<sub>i+1</sub> =



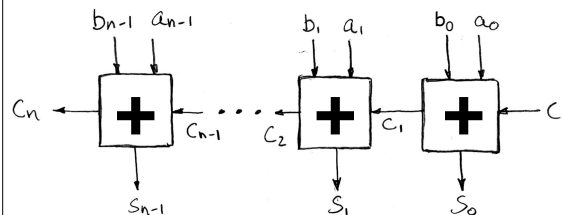
**Adder/Subtractor – One-bit adder (2/2)...**



s<sub>i</sub> = XOR(a<sub>i</sub>, b<sub>i</sub>, c<sub>i</sub>)  
 c<sub>i+1</sub> = MAJ(a<sub>i</sub>, b<sub>i</sub>, c<sub>i</sub>) = a<sub>i</sub>b<sub>i</sub> + a<sub>i</sub>c<sub>i</sub> + b<sub>i</sub>c<sub>i</sub>

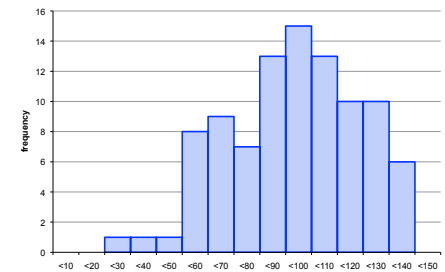


**N 1-bit adders ⇒ 1 N-bit adder**

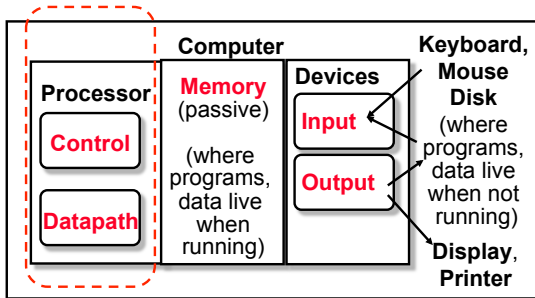


**Administrivia**

- Midterm handed back today



## Five Components of a Computer



CS61CL L09 Single Cycle CPU Design (10)

Huddleston, Summer 2009 © UCB

## The CPU

- **Processor (CPU):** the active part of the computer, which does all the work (data manipulation and decision-making)
- **Datapath:** portion of the processor which contains hardware necessary to perform operations required by the processor (the brawn)
- **Control:** portion of the processor (also in hardware) which tells the datapath what needs to be done (the brain)



CS61CL L09 Single Cycle CPU Design (11)

Huddleston, Summer 2009 © UCB

## Stages of the Datapath : Overview

- **Problem:** a single, atomic block which “executes an instruction” (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient
- **Solution:** break up the process of “executing an instruction” into **stages**, and then connect the stages to create the whole datapath
  - smaller stages are easier to design
  - easy to optimize (change) one stage without touching the others



CS61CL L09 Single Cycle CPU Design (12)

Huddleston, Summer 2009 © UCB

## Stages of the Datapath (1/5)

- There is a wide variety of MIPS instructions: so what general steps do they have in common?
- **Stage 1: Instruction Fetch**
  - no matter what the instruction, the 32-bit instruction must first be fetched from memory (the cache-memory hierarchy)
  - also, this is where we **Increment PC** (that is,  $PC = PC + 4$ , to point to the next instruction: byte addressing so + 4)



CS61CL L09 Single Cycle CPU Design (13)

Huddleston, Summer 2009 © UCB

## Stages of the Datapath (2/5)

- **Stage 2: Instruction Decode**
  - upon fetching the instruction, we next gather data from the fields (decode all necessary instruction data)
  - first, read the *opcode* to determine instruction type and field lengths
  - second, read in data from all necessary registers
    - for *add*, read two registers
    - for *addi*, read one register
    - for *j al*, no reads necessary



CS61CL L09 Single Cycle CPU Design (14)

Huddleston, Summer 2009 © UCB

## Stages of the Datapath (3/5)

- **Stage 3: ALU (Arithmetic-Logic Unit)**
  - the real work of most instructions is done here: arithmetic (+, -, \*, /), shifting, logic (&, |), comparisons (*slt*)
  - what about loads and stores?
    - `lw $t0, 40($t1)`
    - the address we are accessing in memory = the value in *\$t1* PLUS the value 40
    - so we do this addition in this stage



CS61CL L09 Single Cycle CPU Design (15)

Huddleston, Summer 2009 © UCB

## Stages of the Datapath (4/5)

- **Stage 4: Memory Access**
  - actually only the load and store instructions do anything during this stage; the others remain idle during this stage or skip it all together
  - since these instructions have a unique step, we need this extra stage to account for them
  - as a result of the cache system, this stage is expected to be fast



CS61CL L09 Single Cycle CPU Design (16)

Huddleston, Summer 2009 © UCB

## Stages of the Datapath (5/5)

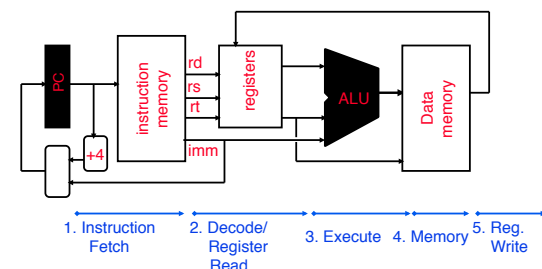
- **Stage 5: Register Write**
  - most instructions write the result of some computation into a register
  - examples: arithmetic, logical, shifts, loads, *slt*
  - what about stores, branches, jumps?
    - don't write anything into a register at the end
    - these remain idle during this fifth stage or skip it all together



CS61CL L09 Single Cycle CPU Design (17)

Huddleston, Summer 2009 © UCB

## Generic Steps of Datapath



CS61CL L09 Single Cycle CPU Design (18)

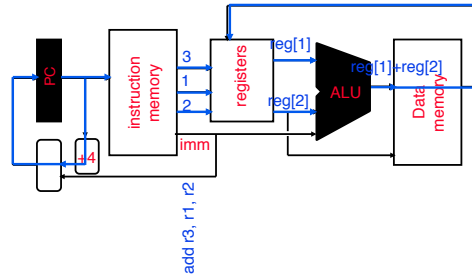
Huddleston, Summer 2009 © UCB

## Datapath Walkthroughs (1/3)

- `add $r3,$r1,$r2 # r3 = r1+r2`
  - Stage 1: fetch this instruction, inc. PC
  - Stage 2: decode to find it's an add, then read registers `$r1` and `$r2`
  - Stage 3: add the two values retrieved in Stage 2
  - Stage 4: idle (nothing to write to memory)
  - Stage 5: write result of Stage 3 into register `$r3`



## Example: add Instruction

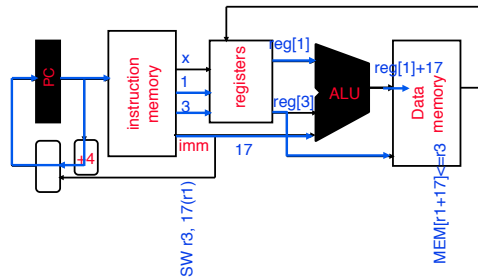


## Datapath Walkthroughs (3/3)

- `sw $r3, 17($r1)`
  - Stage 1: fetch this instruction, inc. PC
  - Stage 2: decode to find it's a `sw`, then read registers `$r1` and `$r3`
  - Stage 3: add 17 to value in register `$r1` (retrieved in Stage 2)
  - Stage 4: write value in register `$r3` (retrieved in Stage 2) into memory address computed in Stage 3
  - Stage 5: idle (nothing to write into a register)



## Example: sw Instruction



## Why Five Stages? (1/2)

- Could we have a different number of stages?
  - Yes, and other architectures do
- So why does MIPS have five if instructions tend to idle for at least one stage?
  - The five stages are the union of all the operations needed by all the instructions.
  - There is one type of instruction that uses all five stages: the **load**

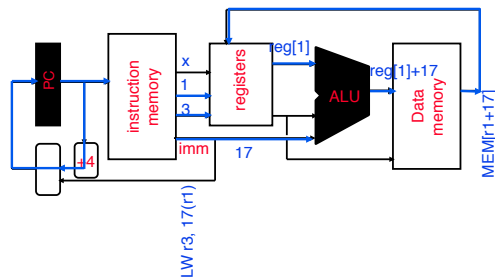


## Why Five Stages? (2/2)

- `lw $r3, 17($r1)`
  - Stage 1: fetch this instruction, inc. PC
  - Stage 2: decode to find it's a `lw`, then read register `$r1`
  - Stage 3: add 17 to value in register `$r1` (retrieved in Stage 2)
  - Stage 4: read value from memory address compute in Stage 3
  - Stage 5: write value found in Stage 4 into register `$r3`

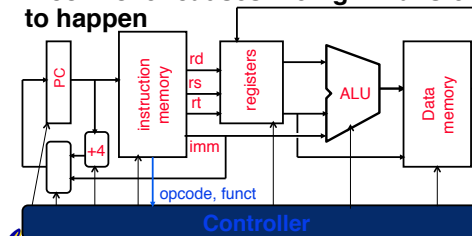


## Example: lw Instruction



## Datapath Summary

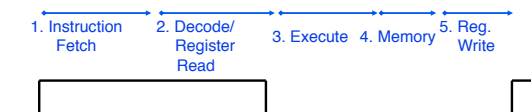
- The datapath based on data transfers required to perform instructions
- A controller causes the right transfers to happen



## CPU clocking (1/2)

For each instruction, how do we control the flow of information through the datapath?

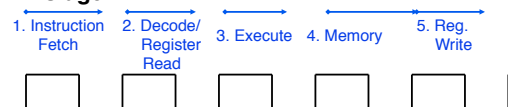
- Single Cycle CPU: All stages of an instruction are completed within one long clock cycle.
  - The clock cycle is made sufficient long to allow each instruction to complete all stages without interruption and within one cycle.



## CPU clocking (2/2) For each instruction, how do we control the flow of information through the datapath?

### Multiple-cycle CPU: Only one stage of instruction per clock cycle.

- The clock is made as long as the **slowest stage**.



- Several significant advantages over single cycle execution: Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped).



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

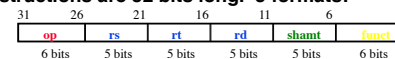
- Analyze instruction set architecture (ISA)
  - ⇒ datapath requirements
  - meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
  - datapath must support each register transfer
- Select set of datapath components and establish clocking methodology
- Assemble datapath meeting requirements
- Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- Assemble the control logic (hard part!)



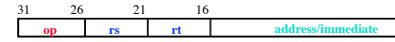
## Review: The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. 3 formats:

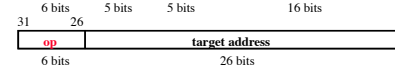
### R-type



### I-type



### J-type



- The different fields are:

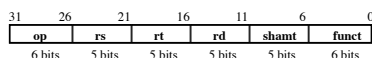
- op: operation ("opcode") of the instruction
- rs, rt, rd: the source and destination register specifiers
- shamt: shift amount
- funct: selects the variant of the operation in the "op" field
- address / immediate: address offset or immediate value
- target address: target address of jump instruction



## The MIPS-lite Subset for today

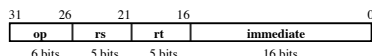
### ADDU and SUBU

- addu rd,rs,rt
- subu rd,rs,rt



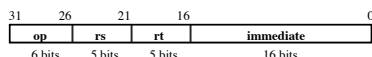
### OR Immediate:

- ori rt,rs,imm16



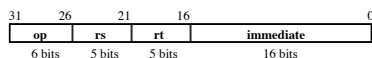
### LOAD and STORE Word

- lw rt,rs,imm16
- sw rt,rs,imm16



### BRANCH:

- beq rs,rt,imm16



## ALU Needs for MIPS-lite + Rest of MIPS

### Addition, subtraction, logical OR, ==:

```
ADDU    R[rd] = R[rs] + R[rt]; ...
SUBU    R[rd] = R[rs] - R[rt]; ...
ORI     R[rt] = R[rs] | zero_ext(Imm16) ...
BEQ     if ( R[rs] == R[rt] ) ...
```

### Test to see if output == 0 for any ALU operation gives == test. How?

### P&H also adds AND, Set Less Than (1 if A < B, 0 otherwise)



ALU follows chap 5

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

- Analyze instruction set architecture (ISA)
  - ⇒ datapath requirements
  - meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
  - datapath must support each register transfer
- Select set of datapath components and establish clocking methodology
- Assemble datapath meeting requirements
- Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- Assemble the control logic (hard part!)



## What Hardware Is Needed? (1/2)

### PC: a register which keeps track of memory addr of the next instruction

### General Purpose Registers

- used in Stages 2 (Read) and 5 (Write)
- MIPS has 32 of these

### Memory

- used in Stages 1 (Fetch) and 4 (R/W)
- cache system makes these two stages as fast as the others, on average



## What Hardware Is Needed? (2/2)

### ALU

- used in Stage 3
- something that performs all necessary functions: arithmetic, logicals, etc.
- we'll design details later

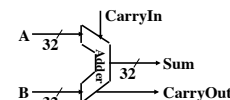
### Miscellaneous Registers

- In implementations with only one stage per clock cycle, registers are inserted between stages to hold intermediate data and control signals as they travels from stage to stage.
- Note: Register is a general purpose term meaning something that stores bits. Not all registers are in the "register file".

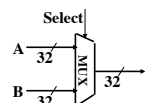


## Combinational Logic Elements (Building Blocks)

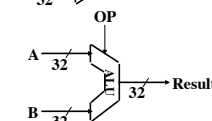
### Adder



### MUX



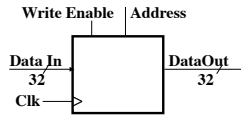
### ALU



## Storage Element: Idealized Memory

### Memory (idealized)

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



### Memory word is found 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)

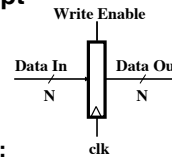
- The CLK input is a factor ONLY during write operation
- During read operation, behaves as a combinational logic block:
  - Address valid  $\Rightarrow$  Data Out valid after "access time."



## Storage Element: Register (Building Block)

### Similar to D Flip Flop except

- N-bit input and output
- Write Enable input



### Write Enable:

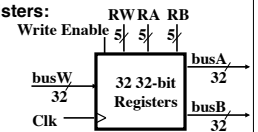
- negated (or deasserted) (0): Data Out will not change
- asserted (1): Data Out will become Data In on positive edge of clock



## Storage Element: Register File

### Register File consists of 32 registers:

- Two 32-bit output buses: 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  $\Rightarrow$  busA or busB valid after "access time."



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

### 1. Analyze instruction set architecture (ISA)

$\Rightarrow$  datapath requirements

- meaning of each instruction is given by the register transfers
- datapath must include storage element for ISA registers
- datapath must support each register transfer

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

### 3. Assemble datapath meeting requirements

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

### 5. Assemble the control logic (hard part!)



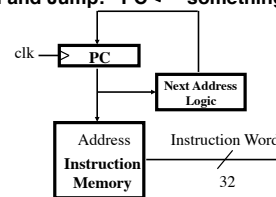
## Overview of the Instruction Fetch Unit

### The common RTL operations

#### Fetch the Instruction: mem[PC]

#### Update the program counter:

- Sequential Code:  $PC \leftarrow PC + 4$
- Branch and Jump:  $PC \leftarrow$  "something else"

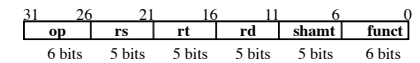


## Add & Subtract

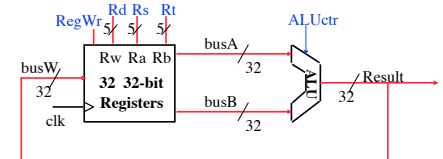
### $R[rd] = R[rs] \text{ op } R[rt]$ Ex.: addU rd,rs,rt

- Ra, Rb, and Rw come from instruction's Rs, Rt, and Rd fields

- ALUctr and RegWr: control logic after decoding the instruction

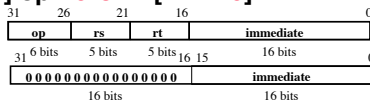


### ... Already defined the register file & ALU

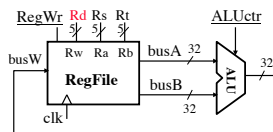


## Logical Operations with Immediate

### $R[rt] = R[rs] \text{ op } \text{ZeroExt}[\text{imm16}]$

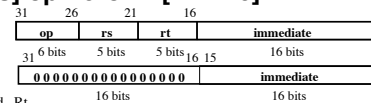


But we're writing to Rt register??

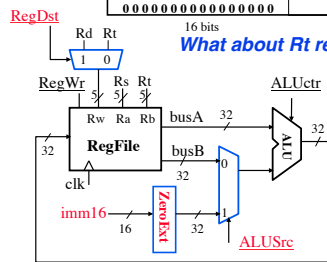


## Logical Operations with Immediate

### $R[rt] = R[rs] \text{ op } \text{ZeroExt}[\text{imm16}]$



What about Rt register read??



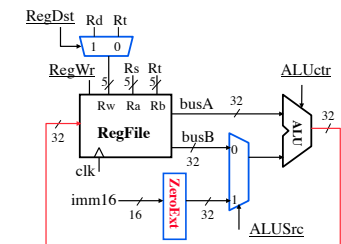
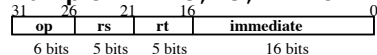
• Already defined 32-bit MUX; Zero Ext?



## Load Operations

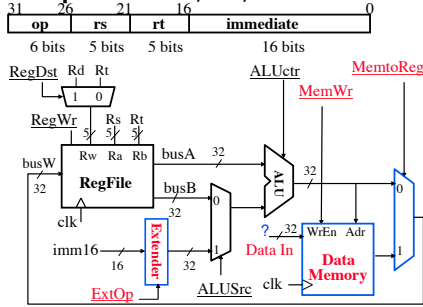
### $R[rt] = \text{Mem}[R[rs]] + \text{SignExt}[\text{imm16}]$

Example: lw rt,rs,imm16



## Load Operations

- $R[rt] = Mem[R[rs] + SignExt[imm16]]$
- Example: `lw rt, rs, imm16`

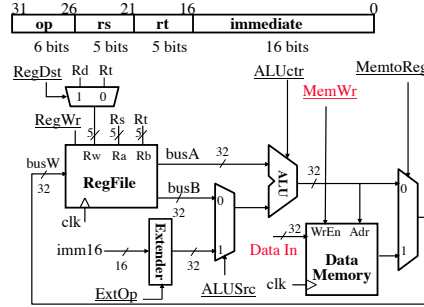


CS61CL L09 Single Cycle CPU Design (46)

Huddleston, Summer 2009 © UCB

## Store Operations

- $Mem[R[rs] + SignExt[imm16]] = R[rt]$
- Ex.: `sw rt, rs, imm16`

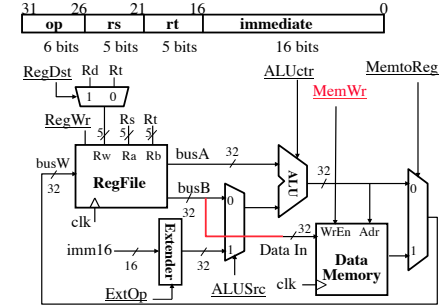


CS61CL L09 Single Cycle CPU Design (47)

Huddleston, Summer 2009 © UCB

## Store Operations

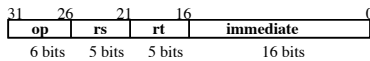
- $Mem[R[rs] + SignExt[imm16]] = R[rt]$
- Ex.: `sw rt, rs, imm16`



CS61CL L09 Single Cycle CPU Design (48)

Huddleston, Summer 2009 © UCB

## The Branch Instruction



`beq rs, rt, imm16`

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

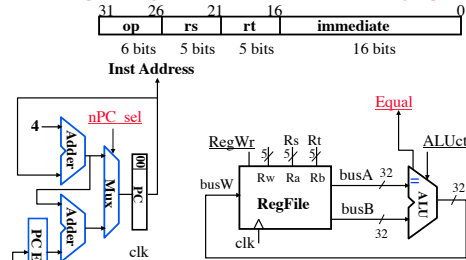


CS61CL L09 Single Cycle CPU Design (49)

Huddleston, Summer 2009 © UCB

## Datapath for Branch Operations

- `beq rs, rt, imm16`
- Datapath generates condition (equal)

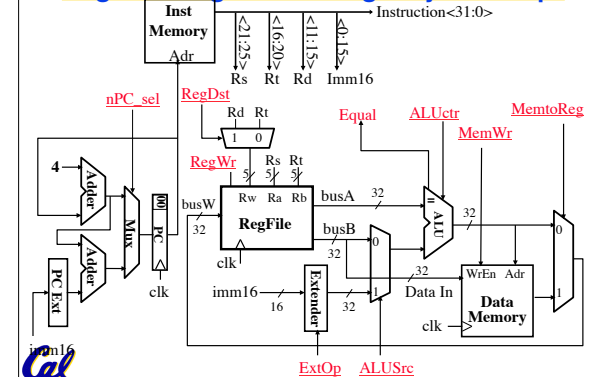


CS61CL L09 Single Cycle CPU Design (50)

Huddleston, Summer 2009 © UCB

Already have mux, adder, need special sign extender for PC, need equal compare (sub?)

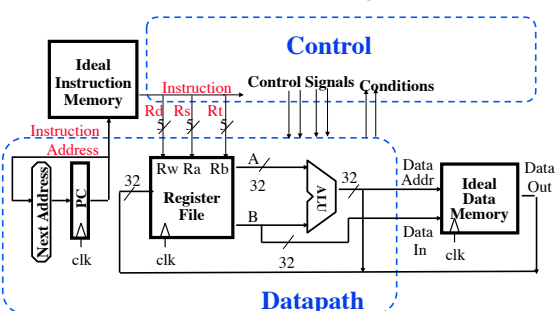
## Putting it All Together: A Single Cycle Datapath



CS61CL L09 Single Cycle CPU Design (51)

Huddleston, Summer 2009 © UCB

## An Abstract View of the Implementation

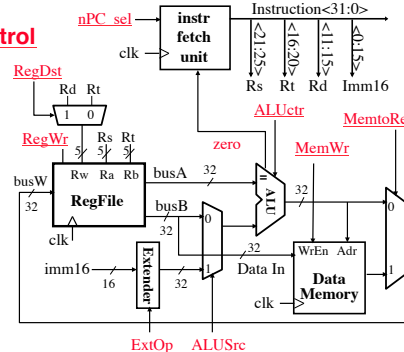


CS61CL L09 Single Cycle CPU Design (52)

Huddleston, Summer 2009 © UCB

## Summary: A Single Cycle Datapath

- We have everything except control signals



CS61CL L09 Single Cycle CPU Design (53)

Huddleston, Summer 2009 © UCB

## “And In conclusion...”

- N-bit adder-subtractor done using N-bit adders with XOR gates on input
  - XOR serves as conditional inverter
- CPU design involves Datapath, Control
  - Datapath in MIPS involves 5 CPU stages
    1. Instruction Fetch
    2. Instruction Decode & Register Read
    3. ALU (Execute)
    4. Memory
    5. Register Write



CS61CL L09 Single Cycle CPU Design (54)

Huddleston, Summer 2009 © UCB



## Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

# Bonus



CS61CL L09 Single Cycle CPU Design (55)

Huddleston, Summer 2009 © UCB

## Review of Timing Terms

- **Clock (CLK)** - steady square wave that synchronizes system
- **Setup Time** - when the input must be stable before the rising edge of the CLK
- **Hold Time** - when the input must be stable after the rising edge of the CLK
- **“CLK-to-Q” Delay** - how long it takes the output to change, measured from the rising edge
- **Flip-flop** - one bit of state that samples every rising edge of the CLK
- **Register** - several bits of state that samples on rising edge of CLK or on LOAD

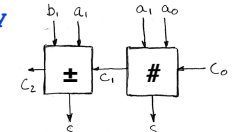


CS61CL L09 Single Cycle CPU Design (56)

Huddleston, Summer 2009 © UCB

## What about overflow?

- Consider a 2-bit signed # & overflow:
  - 10 = -2 + -2 or -1
  - 11 = -1 + -2 only
  - 00 = 0 NOTHING!
  - 01 = 1 + 1 only



### Highest adder

- $C_1 = \text{Carry-in} = C_{in}$ ,  $C_2 = \text{Carry-out} = C_{out}$
- No  $C_{out}$  or  $C_{in} \Rightarrow$  NO overflow!

What  $C_{in}$  and  $C_{out} \Rightarrow$  NO overflow!

op?

- $C_{in}$  but no  $C_{out} \Rightarrow$  A,B both > 0, overflow!
- $C_{out}$  but no  $C_{in} \Rightarrow$  A,B both < 0, overflow!



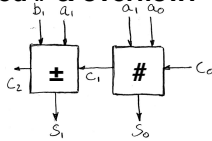
CS61CL L09 Single Cycle CPU Design (57)

Huddleston, Summer 2009 © UCB

## What about overflow?

- Consider a 2-bit signed # & overflow:

10 = -2  
11 = -1  
00 = 0  
01 = 1



### Overflows when...

- $C_{in}$ , but no  $C_{out} \Rightarrow$  A,B both > 0, overflow!
- $C_{out}$ , but no  $C_{in} \Rightarrow$  A,B both < 0, overflow!

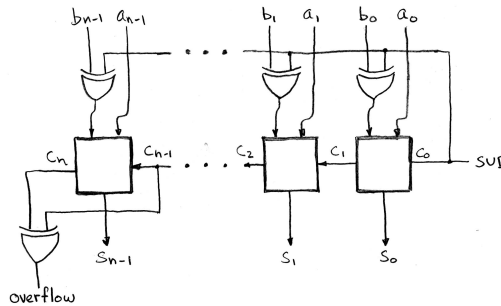
$$\text{overflow} = C_n \text{ XOR } C_{n-1}$$



CS61CL L09 Single Cycle CPU Design (58)

Huddleston, Summer 2009 © UCB

## Extremely Clever Subtractor



CS61CL L09 Single Cycle CPU Design (59)

Huddleston, Summer 2009 © UCB

## Datapath Walkthroughs (2/3)

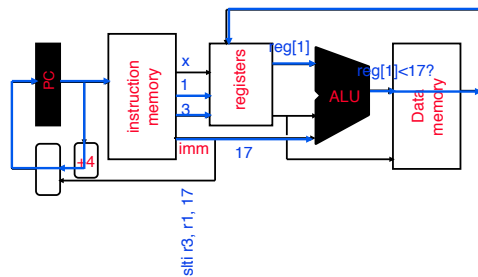
- `s1ti $r3, $r1, 17`
- Stage 1: fetch this instruction, inc. PC
- Stage 2: decode to find it's an `s1ti`, then read register `$r1`
- Stage 3: compare value retrieved in Stage 2 with the integer 17
- Stage 4: idle
- Stage 5: write the result of Stage 3 in register `$r3`



CS61CL L09 Single Cycle CPU Design (60)

Huddleston, Summer 2009 © UCB

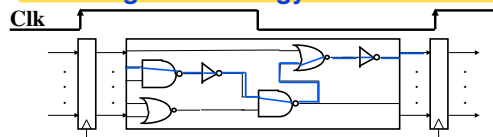
## Example: `s1ti` Instruction



CS61CL L09 Single Cycle CPU Design (61)

Huddleston, Summer 2009 © UCB

## Clocking Methodology



- Storage elements clocked by same edge
- Being physical devices, flip-flops (FF) and combinational logic have some delays
  - Gates: delay from input change to output change
  - Signals at FF D input must be stable before active clock edge to allow signal to travel within the FF (set-up time), and we have the usual clock-to-Q delay
- **“Critical path”** (longest path through logic) determines length of clock period



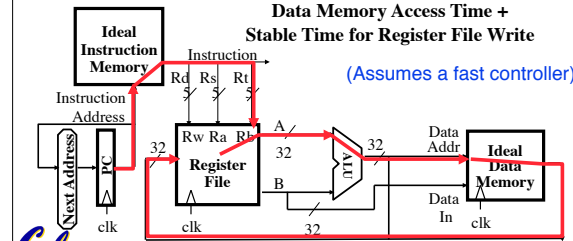
CS61CL L09 Single Cycle CPU Design (62)

Huddleston, Summer 2009 © UCB

## An Abstract View of the Critical Path

Critical Path (Load Instruction) =  
Delay clock through PC (FFs) +  
Instruction Memory's Access Time +  
Register File's Access Time, +  
ALU to Perform a 32-bit Add +  
Data Memory Access Time +  
Stable Time for Register File Write

(Assumes a fast controller)



CS61CL L09 Single Cycle CPU Design (63)

Huddleston, Summer 2009 © UCB