

CS 61C: Great Ideas in Computer  
Architecture (Machine Structures)  
Single-Cycle CPU  
*Datapath Control Part 1*

Instructors:

Krste Asanovic & Vladimir Stojanovic  
<http://inst.eecs.berkeley.edu/~cs61c/>

### Review

- Timing constraints for Finite State Machines
  - Setup time, Hold Time, Clock to Q time
- Use muxes to select among inputs
  - S control bits selects from 2S inputs
  - Each input can be n-bits wide, indep of S
  - Can implement muxes hierarchically
- ALU can be implemented using a mux
  - Coupled with basic block elements

### How to design Adder/Subtractor?

- Truth-table, then determine canonical form, then minimize and implement as we've seen before
- Look at breaking the problem down into smaller pieces that we can cascade or hierarchically layer

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

	a <sub>3</sub>	a <sub>2</sub>	a <sub>1</sub>	a <sub>0</sub>		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
	s <sub>3</sub>	s <sub>2</sub>	s <sub>1</sub>	s <sub>0</sub>		1	1	0	1

$$s_0 =$$

$$c_1 =$$

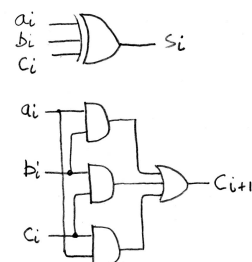
### Adder/Subtractor – One-bit adder (1/2)...

	a <sub>3</sub>	a <sub>2</sub>	a <sub>1</sub>	a <sub>0</sub>		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

$$s_i =$$

$$c_{i+1} =$$

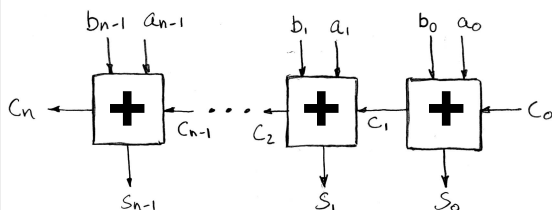
### Adder/Subtractor – One-bit adder (2/2)



$$s_i = \text{XOR}(a_i, b_i, c_i)$$

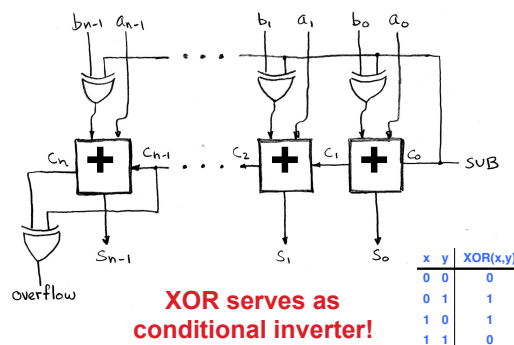
$$c_{i+1} = \text{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i$$

### N 1-bit adders $\Rightarrow$ 1 N-bit adder



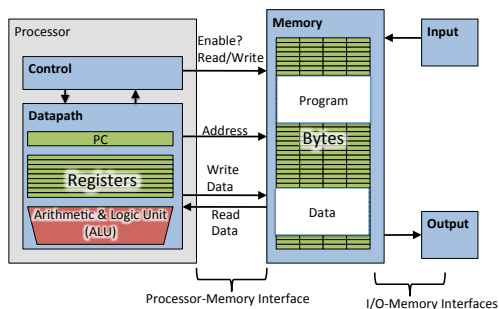
What about overflow?  
Overflow =  $c_n$ ?

### Extremely Clever Subtractor



XOR serves as  
conditional inverter!

### Components of a Computer



### The CPU

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

### Five Stages of Instruction Execution

- Stage 1: Instruction Fetch
- Stage 2: Instruction Decode
- Stage 3: ALU (Arithmetic-Logic Unit)
- Stage 4: Memory Access
- Stage 5: Register Write

### Stages of Execution (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 word 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)

### Stages of Execution (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 jal, no reads necessary

### Stages of Execution (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

### Stages of Execution (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

### Stages of Execution (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

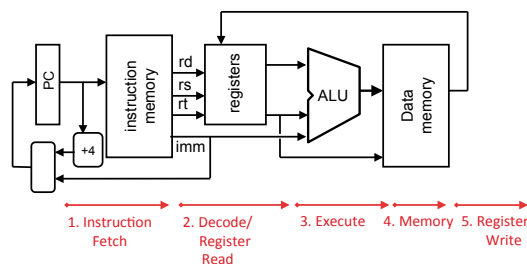
### Administrivia

2/24/15

Fall 2011 – Lecture #28

17

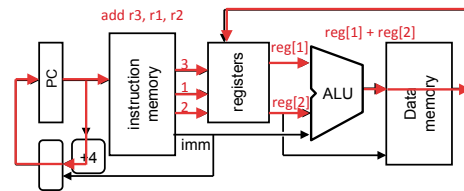
### Stages of Execution on Datapath



### Datapath Walkthroughs (1/3)

- add \$r3,\$r1,\$r2 #  $r3 = r1 + r2$ 
  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine it is 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

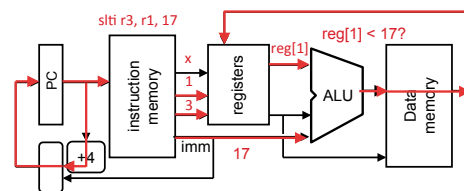
### Example: add Instruction



### Datapath Walkthroughs (2/3)

- slti \$r3,\$r1,17  
# if ( $r1 < 17$ )  $r3 = 1$  else  $r3 = 0$ 
  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine it is an slti, 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 (1 if reg source was less than signed immediate, 0 otherwise) into register \$r3

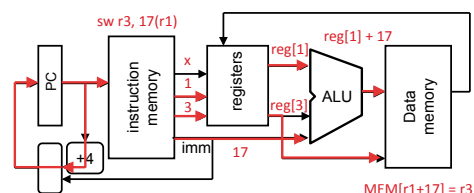
### Example: slti Instruction



### Datapath Walkthroughs (3/3)

- sw \$r3,17(\$r1) #  $Mem[r1+17] = r3$ 
  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine it is a sw, then read registers \$r1 and \$r3
  - Stage 3: add 17 to value in register \$r1 (retrieved in Stage 2) to compute address
  - 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



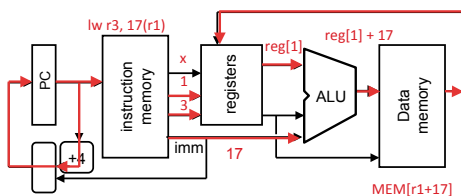
### Why Five Stages? (1/2)

- Could we have a different number of stages?
  - Yes, other ISAs have different natural number of stages
- Why does MIPS have five if instructions tend to idle for at least one stage?
  - Five stages are the union of all the operations needed by all the instructions.
  - One instruction uses all five stages: the load

### Why Five Stages? (2/2)

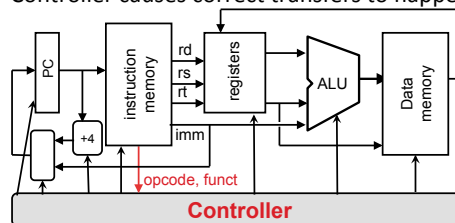
- `lw $r3, 17($r1) # r3=Mem[r1+17]`
  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine it is a lw, then read register \$r1
  - Stage 3: add 17 to value in register \$r1 (retrieved in Stage 2)
  - Stage 4: read value from memory address computed in Stage 3
  - Stage 5: write value read in Stage 4 into register \$r3

### Example: lw Instruction



### Datapath and Control

- Datapath designed to support data transfers required by instructions
- Controller causes correct transfers to happen



### In the News

- At ISSCC 2015 in San Francisco yesterday, latest IBM mainframe chip details
- z13 designed in 22nm SOI technology with **seventeen** metal layers, 4 billion transistors/chip
- 8 cores/chip, with 2MB L2 cache, 64MB L3 cache, and 480MB L4 cache.
- 5GHz clock rate, 6 instructions per cycle, 2 threads/core
- Up to 24 processor chips in shared memory node

2/24/15

Fall 2015 – Lecture #28

29

Register Now:  
 Berkeley – IAP Workshop on the Future of Cloud Technologies  
 Friday, February 27, 2015

☐ POSTERS: \$500 Prizes for Best Undergrad & Best Grad  
☐ CAREER FAIR: During Lunch/Breaks, Bring Your Resume  
<http://www.industry-academia.org/event-berkeley-cloud-workshop.html>

**Quotes from the MIT Workshop**

"I was excited by the turnout and the breadth of the speakers." - Prof. Matei Zaharia, CTO Databricks

"In a nutshell, the workshop is an excellent opportunity for students, faculties and people from industry to share ideas." - Po-An Tsai, PhD Student, MIT CSAIL, Best Poster Award

## Processor Design: 5 steps

Step 1: Analyze instruction set to determine datapath requirements

- Meaning of each instruction is given by register transfers
- Datapath must include storage element for ISA registers
- Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

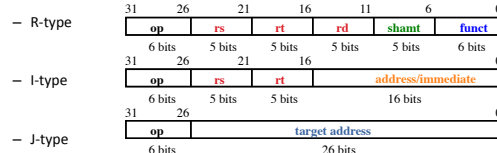
Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic

## The MIPS Instruction Formats

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



- 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

- 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`

## Register Transfer Level (RTL)

- Colloquially called "Register Transfer Language"
- RTL gives the meaning of the instructions
- All start by fetching the instruction itself

```
{op, rs, rt, rd, shamt, funct} ← MEM[ PC ]
{op, rs, rt, Imm16} ← MEM[ PC ]

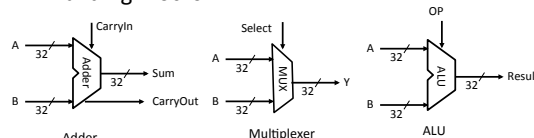
Inst Register Transfers
ADDU R[rd] ← R[rs] + R[rt]; PC ← PC + 4
SUBU R[rd] ← R[rs] - R[rt]; PC ← PC + 4
ORI R[rt] ← R[rs] | zero_ext(Imm16); PC ← PC + 4
LOAD R[rt] ← MEM[ R[rs] + sign_ext(Imm16) ]; PC ← PC + 4
STORE MEM[ R[rs] + sign_ext(Imm16) ] ← R[rt]; PC ← PC + 4
BEQ if ( R[rs] == R[rt] )
    PC ← PC + 4 + {sign_ext(Imm16), 2'b00}
    else PC ← PC + 4
```

## Step 1: Requirements of the Instruction Set

- Memory (MEM)
  - Instructions & data (will use one for each)
- Registers (R: 32, 32-bit wide registers)
  - Read RS
  - Read RT
  - Write RT or RD
- Program Counter (PC)
- Extender (sign/zero extend)
- Add/Sub/OR/etc unit for operation on register(s) or extended immediate (ALU)
- Add 4 (+ maybe extended immediate) to PC
- Compare registers?

## Step 2: Components of the Datapath

- Combinational Elements
- Storage Elements + Clocking Methodology
- Building Blocks



### 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 Chapter 5

### Storage Element: Idealized Memory

- “Magic” Memory

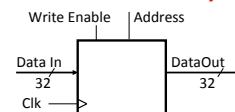
- One input bus: Data In
- One output bus: Data Out

- Memory word is found by:

- For Read: Address selects the word to put on Data Out
- For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus

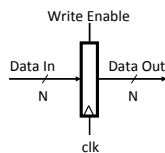
- Clock input (CLK)

- 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



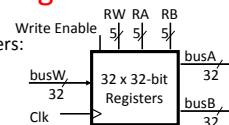
### Storage Element: Register File

- Register File consists of 32 registers:

- Two 32-bit output busses: busA and busB
- One 32-bit input bus: busW

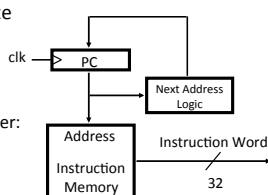
- Register is selected by:

- 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)
  - 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.”



### Step 3a: Instruction Fetch Unit

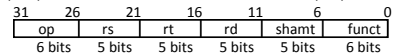
- Register Transfer Requirements  $\Rightarrow$  Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation
- Common RTL operations
  - Fetch the Instruction: `mem[PC]`
  - Update the program counter:
    - Sequential Code: `PC  $\leftarrow$  PC + 4`
    - Branch and Jump: `PC  $\leftarrow$  “something else”`



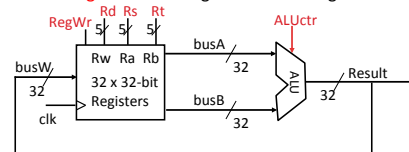
### Step 3b: Add & Subtract

- `R[rd] = R[rs] op R[rt] (addu rd,rs,rt)`

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

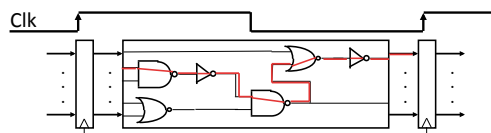


- ALUctr and RegWr: control logic after decoding the instruction



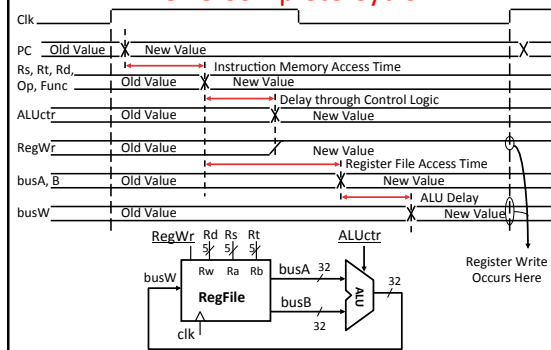
- ... Already defined the register file & ALU

### Clocking Methodology

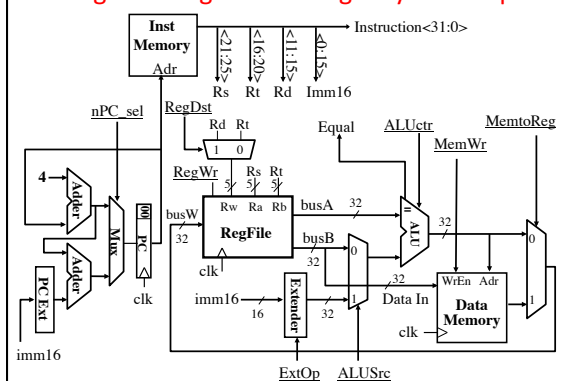


- Storage elements clocked by same edge
- Flip-flops (FFs) and combinational logic have some delays
  - Gates: delay from input change to output change
  - Signals at FF D input must be stable before active clock edge to allow signal to travel within the FF (set-up time), and we have the usual clock-to-Q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

### Register-Register Timing: One Complete Cycle



### Putting it All Together: A Single Cycle Datapath



### Peer Instruction

- Our ALU is a synchronous device
- We should use the main ALU to compute  $PC = PC + 4$
- The ALU is inactive for memory reads or writes.

Option	1	2	3
A	F	F	F
A	T	T	T
B	F	T	T
C	F	T	T
C	T	F	F
D	T	F	T
E	T	T	F
E	F	F	T

### Processor Design: 3 of 5 steps

- Step 1: Analyze instruction set to determine datapath requirements
  - Meaning of each instruction is given by register transfers
  - Datapath must include storage element for ISA registers
  - Datapath must support each register transfer
- Step 2: Select set of datapath components & establish clock methodology
- Step 3: Assemble datapath components that meet the requirements
- Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer
- Step 5: Assemble the control logic

### In Conclusion

- “Divide and Conquer” to build complex logic blocks from smaller simpler pieces (adder)
- Five stages of MIPS instruction execution
- Mapping instructions to datapath components
- Single long clock cycle per instruction