

## ECE 152 Introduction to Computer Architecture

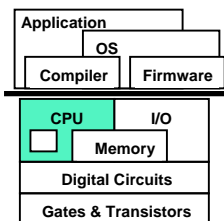
Processor Design: Datapath and Control  
Copyright 2008 Daniel J. Sorin  
Duke University

Slides are derived from work by  
Amir Roth (Penn)  
Spring 2008

## Where We Are in This Course Right Now

- So far:
  - We know what a computer architecture is
  - We know what kinds of instructions it might execute
  - We know how to perform arithmetic and logic in an ALU
- **Now:**
  - We learn how to design a processor in which the ALU is just one component
  - Processor must be able to fetch instructions, decode them, and execute them
  - There are many ways to do this, even for a given ISA
- **Next:**
  - We learn how to use pipelining to get better performance out of this processor

## This Unit: Processor Design



- **Datapath components and timing**
  - Registers and register files
  - Memories (RAMs)
  - Clocking strategies
- Mapping an ISA to a datapath
- Control
  - Single-cycle control
    - RAM/PLA
  - Multi-cycle control
    - RAM/PLA
    - Micro-programmed control
  - Implementing exceptions using control

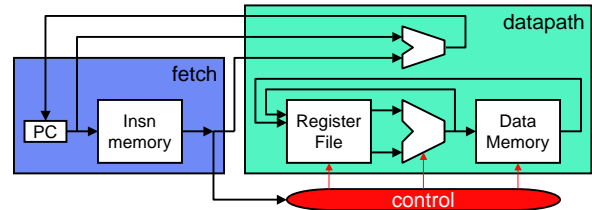
## Readings

- Patterson and Hennessy
  - Chapter 5
- Read this chapter carefully
  - It has many more examples than I can cover in class

## So You Have an ALU...

- **Important reminder:** a processor is just a big finite state machine (FSM) that interprets some ISA
- Start with one instruction
  - `add $3,$2,$4`
  - ALU performs just a small part of execution of instruction
  - You have to read and write registers
  - You have to fetch the instruction to begin with
- What about loads and stores?
  - Need some sort of memory interface
- What about branches?
  - Need some hardware for that, too

## Datapath and Control



- **Datapath:** registers, memories, ALUs (computation)
- **Control:** which registers read/write, which ALU operation
- **Fetch:** get instruction, translate into control
- Processor Cycle: **Fetch** → **Decode** → **Execute**

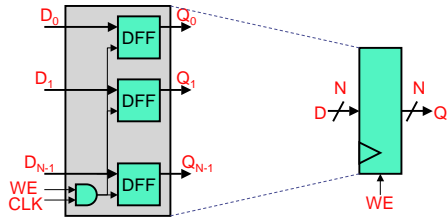
## Building a Processor for an ISA

- Fetch is pretty straightforward
  - Just need a register (called the Program Counter or PC) to hold the next address to fetch from instruction memory
  - Provide address to instruction memory → instruction memory provides instruction at that address
- Let's start with the datapath
  1. Look at ISA
  2. Make sure datapath can implement every instruction

## Datapath for MIPS ISA

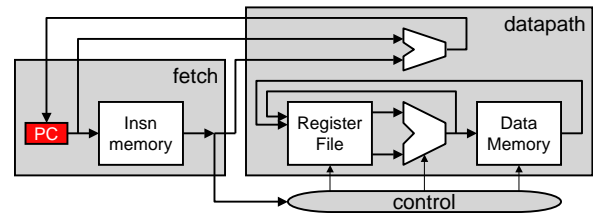
- Consider only the following instructions
  - `add $1,$2,$3`
  - `addi $1,2,$3`
  - `lw $1,4($3)`
  - `sw $1,4($3)`
  - `beq $1,$2,PC_relative_target`
  - `j Absolute_target`
- Why only these?
  - Most other instructions are similar from datapath viewpoint
  - I leave the ones that aren't for you to figure out

## Review (ECE 52 & Project Part1): Register



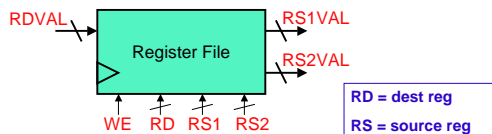
- **Register:** DFF array with shared clock, write-enable (WE)
  - Notice: both a clock and a WE ( $DFF_{WE} = \text{clock} \& \text{register}_{WE}$ )
  - Convention I: clock represented by wedge
  - Convention II: if no WE, DFF is written on every clock

## Uses of Registers



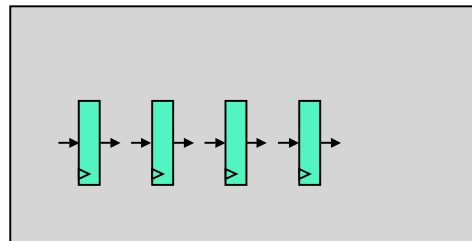
- A single register is good for some things
  - PC: program counter
  - Other things which aren't the ISA registers
    - ICQ: other examples from within the ALU, mult, div?

## What About the ISA Registers?

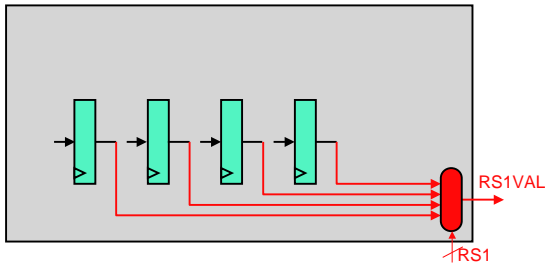


- **Register file:** the ISA ("architectural", "visible") registers
  - Two read "ports" + one write "port"
    - Maximum number of reads/writes in single instruction (R-type)
- **Port:** wires for accessing an array of data
  - Data bus: width of data element (MIPS: 32 bits)
  - Address bus: width of  $\log_2$  number of elements (MIPS: 5 bits)
  - Write enable: if it's a write port
  - M ports = M parallel and independent accesses

## A Register File With Four Registers

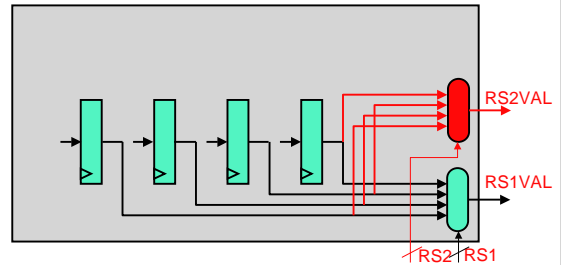


### Add a Read Port for RS1



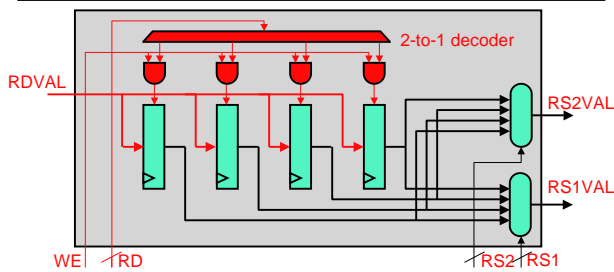
- Output of each register into 4to1 mux (RS1VAL)
  - RS1 is select input of RS1VAL mux

### Add Another Read Port for RS2



- Output of each register into another 4to1 mux (RS2VAL)
  - RS2 is select input of RS2VAL mux

### Add a Write Port for RD

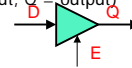


- Input RDVAL into each register
  - Enable only one register's WE: (Decoded RD) & (WE)
- What if we needed two write ports?

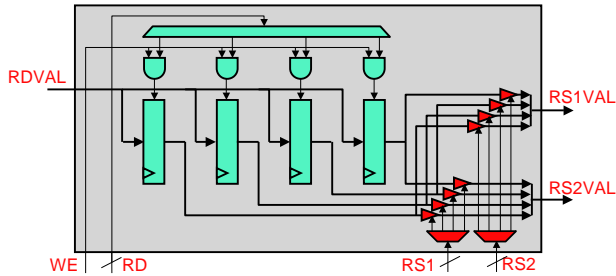
### Another Read Port Implementation

- A read port that uses muxes is fine for 4 registers
  - Not so good for 32 registers (32-to-1 mux is very slow)
- Alternative implementation uses **tri-state buffers**
  - Truth table (E = enable, D = input, Q = output)
 

E	D	Q
1	D	D
0	D	Z
  - **Z**: "high impedance" state, no current flowing
- Mux: connect multiple tri-stated buses to one output bus
- Key: only one input "driving" at any time, all others must be in "Z"
  - Else, all hell breaks loose (electrically)



## Register File With Tri-State Read Ports



© 2008 Daniel J. Sorin  
from Roth

ECE152

17

## Clocking Methodology

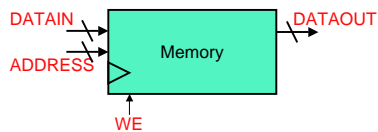
- A single register (e.g., PC)
  - Written on clock edge
  - **Read "continuously"** (new value available after write occurs)
- What about a register file?
  - Written on clock edge
  - **Read on clock edge**
    - On non-writing clock edge (otherwise races would occur)
- We'll walk through clocking examples later
  - This is just a preview of coming attractions

© 2008 Daniel J. Sorin  
from Roth

ECE152

18

## Another Useful Component: Memory



- **Memory**: where instructions and data reside
  - One read/write "port": one access per cycle, either read **or** write
    - One address bus
  - One input data bus for writes, one output data bus for reads
- Actually, a more traditional definition of memory is
  - One input/output data bus
  - No clock → asynchronous "strobe" instead

© 2008 Daniel J. Sorin  
from Roth

ECE152

19

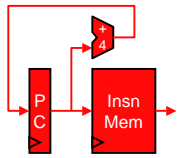
## Let's Build A MIPS-like Datapath

© 2008 Daniel J. Sorin  
from Roth

ECE152

20

## Start With Fetch



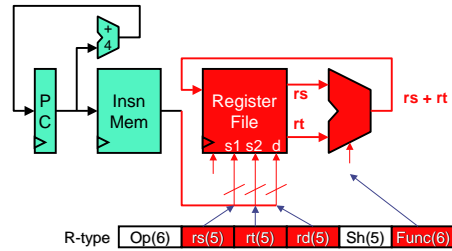
- PC and instruction memory
- A +4 incrementer computes default next instruction PC
  - Why +4 (and not +1)? What will it be for 16-bit Duke 152/16?

© 2008 Daniel J. Sorin  
from Roth

ECE152

21

## First Instruction: add \$rd, \$rs, \$rt



R-type Op(6) rs(5) rt(5) rd(5) Sh(5) Func(6)

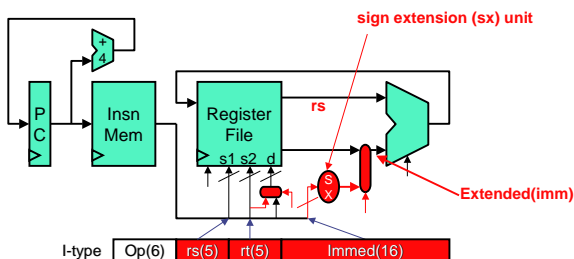
- Add register file and ALU

© 2008 Daniel J. Sorin  
from Roth

ECE152

22

## Second Instruction: addi \$rt, \$rs, imm



I-type Op(6) rs(5) rt(5) Immed(16)

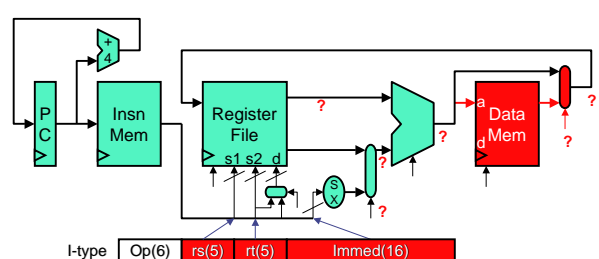
- Destination register can now be either rd or rt
- Add sign extension unit and mux into second ALU input

© 2008 Daniel J. Sorin  
from Roth

ECE152

23

## Third Instruction: lw \$rt, imm(\$rs)



I-type Op(6) rs(5) rt(5) Immed(16)

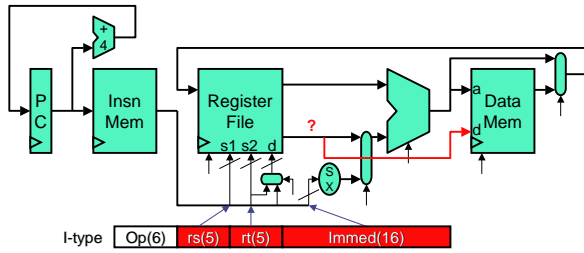
- Add data memory, address is ALU output (rs+imm)
- Add register write data mux to select memory output or ALU output

© 2008 Daniel J. Sorin  
from Roth

ECE152

24

### Fourth Instruction: sw \$rt, imm(\$rs)



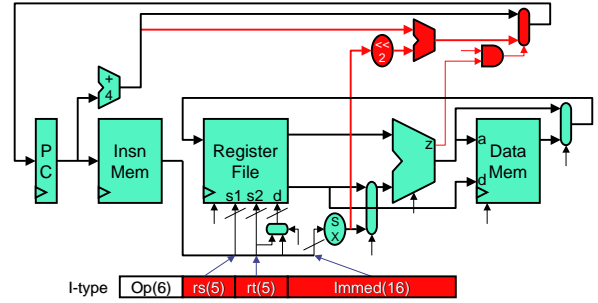
- Add path from second input register to data memory data input
- Disable RegFile's WE signal

© 2008 Daniel J. Sorin  
from Roth

ECE152

25

### Fifth Instruction: beq \$1,\$2,target



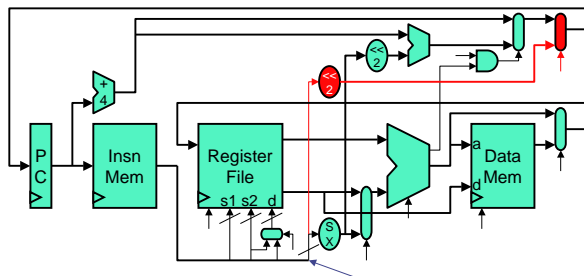
- Add left shift unit (why?) and adder to compute PC-relative branch target
- Add mux to do what?

© 2008 Daniel J. Sorin  
from Roth

ECE152

26

### Sixth Instruction: j



- Add shifter to compute left shift of 26-bit immediate
- Add additional PC input mux for jump target

© 2008 Daniel J. Sorin  
from Roth

ECE152

27

### Seventh, Eighth, Ninth Instructions

- Are these the paths we would need for all instructions?
  - sll \$1,\$2,4**
    - Like an arithmetic operation, but need a shifter too
  - slt \$1,\$2,\$3**
    - Like subtract, but need to write the condition bits, not the result
      - Need zero extension unit for condition bits
      - Need additional input to register write data mux
  - jal absolute\_target**
    - Like a jump, but also need to write PC+4 into \$ra (\$31)
      - Need path from PC+4 adder to register write data mux
      - Need to be able to specify \$31 as an implicit destination
  - jr \$31**
    - Like a jump, but need path from register read to PC write mux

© 2008 Daniel J. Sorin  
from Roth

ECE152

28