

CIS 371 Computer Organization and Design

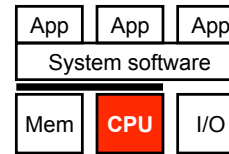
Unit 3: Single-Cycle Datapath

Based on slides by Prof. Amir Roth & Prof. Milo Martin

Readings

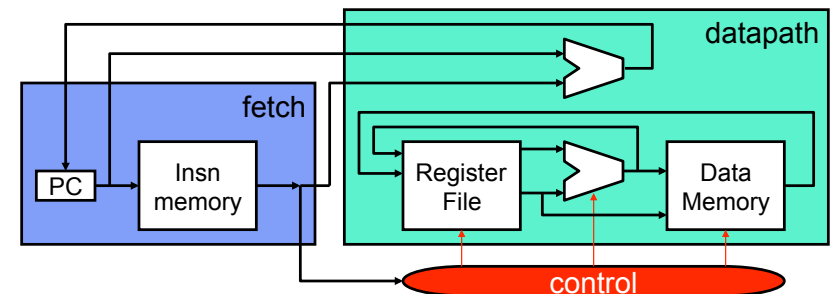
- P&H
 - Sections 4.1 – 4.4

This Unit: Single-Cycle Datapath



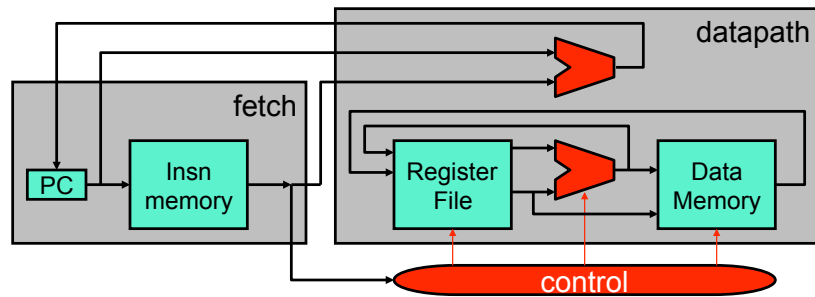
- Datapath storage elements
- MIPS Datapath
- MIPS Control

Motivation: Implementing an ISA



- **Datapath**: performs computation (registers, ALUs, etc.)
 - ISA specific: can implement every insn (single-cycle: in one pass!)
- **Control**: determines which computation is performed
 - Routes data through datapath (which regs, which ALU op)
- **Fetch**: get insn, translate opcode into control
- **Fetch** → **Decode** → **Execute** "cycle"

Two Types of Components

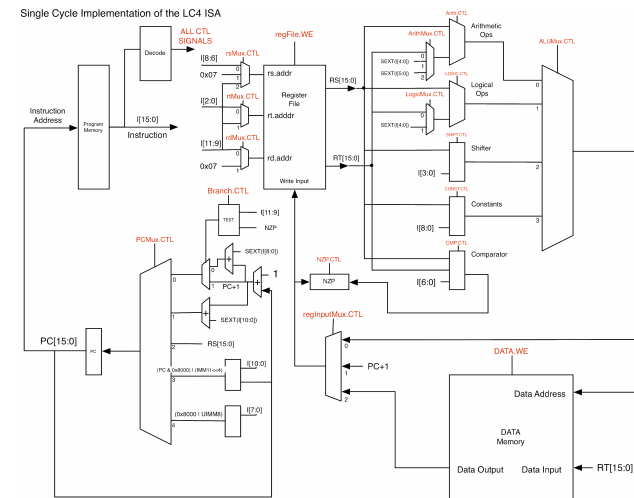


- **Purely combinational:** stateless computation
 - ALUs, muxes, control
 - Arbitrary Boolean functions
- **Combinational/sequential:** storage
 - PC, insn/data memories, register file
 - Internally contain some combinational components

CIS 371 (Martin): Single-Cycle Datapath

5

Example Datapath

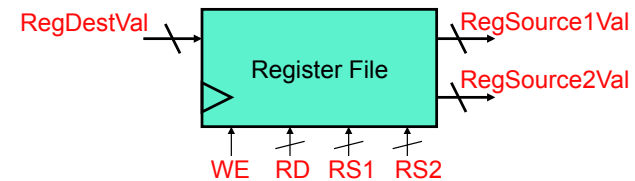


CIS 371 (Martin): Single-Cycle Datapath

6

Datapath Storage Elements

Register File



- **Register file:** M N-bit storage words
 - Multiplexed input/output: data buses write/read "random" word
- **"Port":** set of buses for accessing a random word in array
 - Data bus (N-bits) + address bus ($\log_2 M$ -bits) + optional WE bit
 - P ports = P parallel and independent accesses
- MIPS integer register file
 - 32 32-bit words, two read ports + one write port (why?)

CIS 371 (Martin): Single-Cycle Datapath

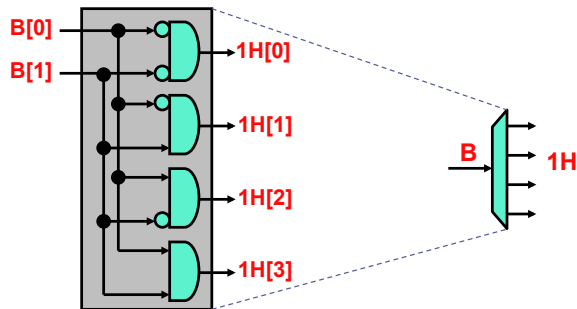
7

CIS 371 (Martin): Single-Cycle Datapath

8

Decoder

- **Decoder**: converts binary integer to "1-hot" representation
 - Binary representation of $0 \dots 2^N - 1$: N bits
 - 1 hot representation of $0 \dots 2^N - 1$: 2^N bits
 - J represented as J^{th} bit 1, all other bits zero
 - Example below: 2-to-4 decoder



Decoder in Verilog (1 of 2)

```
module decoder_2_to_4 (binary_in, onehot_out);
    input [1:0] binary_in;
    output [3:0] onehot_out;
    assign onehot_out[0] = (~binary_in[0] & ~binary_in[1]);
    assign onehot_out[1] = (~binary_in[0] & binary_in[1]);
    assign onehot_out[2] = (binary_in[0] & ~binary_in[1]);
    assign onehot_out[3] = (binary_in[0] & binary_in[1]);
endmodule
```

- Is there a simpler way?

Decoder in Verilog (2 of 2)

```
module decoder_2_to_4 (binary_in, onehot_out);
    input [1:0] binary_in;
    output [3:0] onehot_out;
    assign onehot_out[0] = (binary_in == 2'd0);
    assign onehot_out[1] = (binary_in == 2'd1);
    assign onehot_out[2] = (binary_in == 2'd2);
    assign onehot_out[3] = (binary_in == 2'd3);
endmodule
```

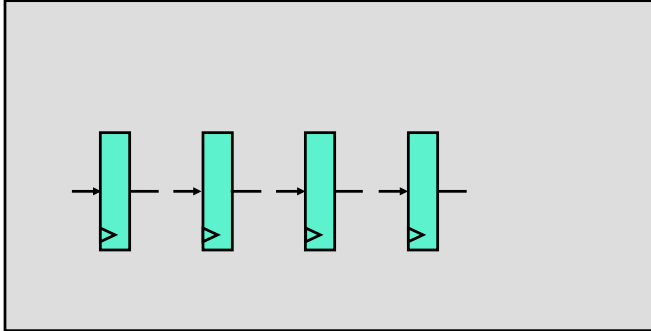
- How is " $a == b$ " implemented for vectors?
 - $|(a \wedge b)$ (this is an "and" reduction of bitwise " $a \text{ xor } b$ ")
 - When one of the inputs to " $==$ " is a constant
 - Simplifies to simpler inverter on bits with "one" in constant
 - Exactly what was on previous slide!

Register File Interface



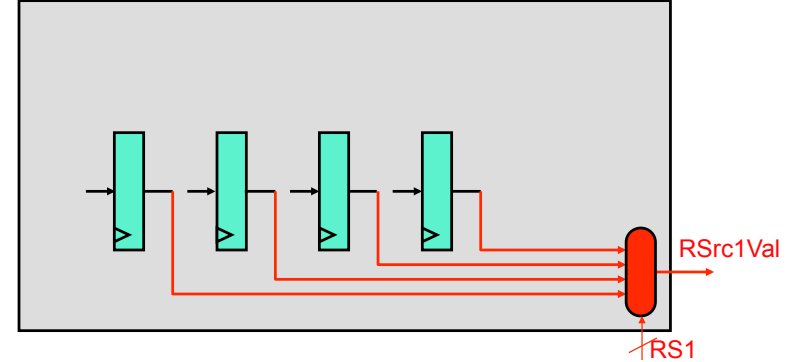
- **Inputs:**
 - RS1, RS2 (reg. sources to read), RD (reg. destination to write)
 - WE (write enable), RDestVal (value to write)
- **Outputs:** RSrc1Val, RSrc2Val (value of RS1 & RS2 registers)

Register File: Four Registers



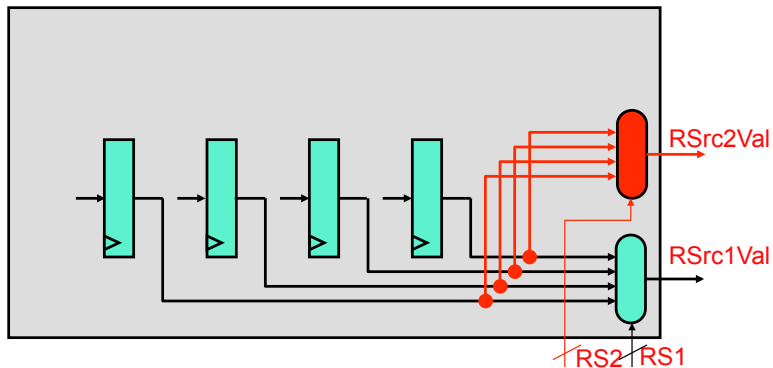
- Register file with four registers

Add a Read Port



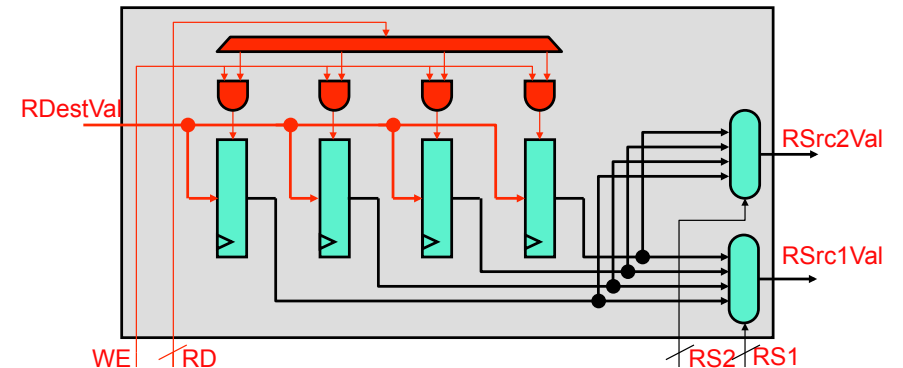
- Output of each register into 4to1 mux (RSrc1Val)
 - RS1 is select input of RSrc1Val mux

Add Another Read Port



- Output of each register into another 4to1 mux (RSrc2Val)
 - RS2 is select input of RSrc2Val mux

Add a Write Port



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

Register File Interface (Verilog)

```
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
  parameter n = 1;
  input [1:0] rs1, rs2, rd;
  input we, rst, clk;
  input [n-1:0] rdval;
  output [n-1:0] rs1val, rs2val;
```

endmodule

- Warning: this code not tested, may contain typos, do not blindly trust!

Register File: Four Registers (Verilog)

```
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
  parameter n = 1;
  input [1:0] rs1, rs2, rd;
  input we, rst, clk;
  input [n-1:0] rdval;
  output [n-1:0] rs1val, rs2val;
  wire [n-1:0] r0v, r1v, r2v, r3v;
```

```
Nbit_reg #(n) r0 (r0v, , , rst, clk);
Nbit_reg #(n) r1 (r1v, , , rst, clk);
Nbit_reg #(n) r2 (r2v, , , rst, clk);
Nbit_reg #(n) r3 (r3v, , , rst, clk);
```

endmodule

- Warning: this code not tested, may contain typos, do not blindly trust!

Add a Read Port (Verilog)

```
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
  parameter n = 1;
  input [1:0] rs1, rs2, rd;
  input we, rst, clk;
  input [n-1:0] rdval;
  output [n-1:0] rs1val, rs2val;
  wire [n-1:0] r0v, r1v, r2v, r3v;
```

```
Nbit_reg #(n) r0 (r0v, , , rst, clk);
Nbit_reg #(n) r1 (r1v, , , rst, clk);
Nbit_reg #(n) r2 (r2v, , , rst, clk);
Nbit_reg #(n) r3 (r3v, , , rst, clk);
Nbit_mux4to1 #(n) mux1 (rs1, r0v, r1v, r2v, r3v, rs1val);
```

endmodule

- Warning: this code not tested, may contain typos, do not blindly trust!

Add Another Read Port (Verilog)

```
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
  parameter n = 1;
  input [1:0] rs1, rs2, rd;
  input we, rst, clk;
  input [n-1:0] rdval;
  output [n-1:0] rs1val, rs2val;
  wire [n-1:0] r0v, r1v, r2v, r3v;
```

```
Nbit_reg #(n) r0 (r0v, , , rst, clk);
Nbit_reg #(n) r1 (r1v, , , rst, clk);
Nbit_reg #(n) r2 (r2v, , , rst, clk);
Nbit_reg #(n) r3 (r3v, , , rst, clk);
Nbit_mux4to1 #(n) mux1 (rs1, r0v, r1v, r2v, r3v, rs1val);
Nbit_mux4to1 #(n) mux2 (rs2, r0v, r1v, r2v, r3v, rs2val);
```

endmodule

- Warning: this code not tested, may contain typos, do not blindly trust!

Add a Write Port (Verilog)

```
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
  parameter n = 1;
  input [1:0] rs1, rs2, rd;
  input we, rst, clk;
  input [n-1:0] rdval;
  output [n-1:0] rs1val, rs2val;
  wire [n-1:0] r0v, r1v, r2v, r3v;
  wire [3:0] rd_select;
  decoder_2_to_4 dec (rd, rd_select);
  Nbit_reg #(n) r0 (r0v, rdval, rd_select[0] & we, rst, clk);
  Nbit_reg #(n) r1 (r1v, rdval, rd_select[1] & we, rst, clk);
  Nbit_reg #(n) r2 (r2v, rdval, rd_select[2] & we, rst, clk);
  Nbit_reg #(n) r3 (r3v, rdval, rd_select[3] & we, rst, clk);
  Nbit_mux4to1 #(n) mux1 (rs1, r0v, r1v, r2v, r3v, rs1val);
  Nbit_mux4to1 #(n) mux2 (rs2, r0v, r1v, r2v, r3v, rs2val);
endmodule
```

- Warning: this code not tested, may contain typos, do not blindly trust!

[intentionally blank]

Final Register File (Verilog)

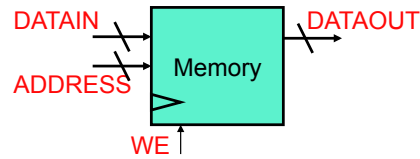
```
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
  parameter n = 1;
  input [1:0] rs1, rs2, rd;
  input we, rst, clk;
  input [n-1:0] rdval;
  output [n-1:0] rs1val, rs2val;
  wire [n-1:0] r0v, r1v, r2v, r3v;

  Nbit_reg #(n) r0 (r0v, rdval, rd == 2`d0 & we, rst, clk);
  Nbit_reg #(n) r1 (r1v, rdval, rd == 2`d1 & we, rst, clk);
  Nbit_reg #(n) r2 (r2v, rdval, rd == 2`d2 & we, rst, clk);
  Nbit_reg #(n) r3 (r3v, rdval, rd == 2`d3 & we, rst, clk);
  Nbit_mux4to1 #(n) mux1 (rs1, r0v, r1v, r2v, r3v, rs1val);
  Nbit_mux4to1 #(n) mux2 (rs2, r0v, r1v, r2v, r3v, rs2val);
endmodule
```

- Warning: this code not tested, may contain typos, do not blindly trust!

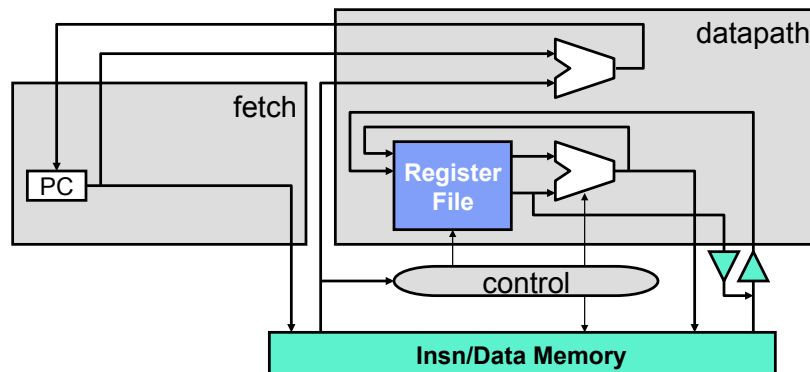
[intentionally blank]

Another Useful Component: Memory



- Register file: M N-bit storage words
 - Few words (< 256), many ports, dedicated read and write ports
- **Memory**: M N-bit storage words, yet not a register file
 - Many words (> 1024), few ports (1, 2), shared read/write ports
- Leads to different implementation choices
 - Lots of circuit tricks and such
 - Larger memories typically only 6 transistors per bit
- In Verilog? We'll give you the code for large memories

Unified vs Split Memory Architecture



- **Unified architecture**: unified insn/data memory
 - LC3, MIPS, every other ISA
- **Harvard architecture**: split insn/data memories
 - LC4

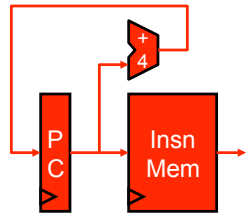
MIPS Datapath

Datapath for MIPS ISA

- MIPS: 32-bit instructions, registers are \$0, \$2... \$31
- Consider only the following instructions

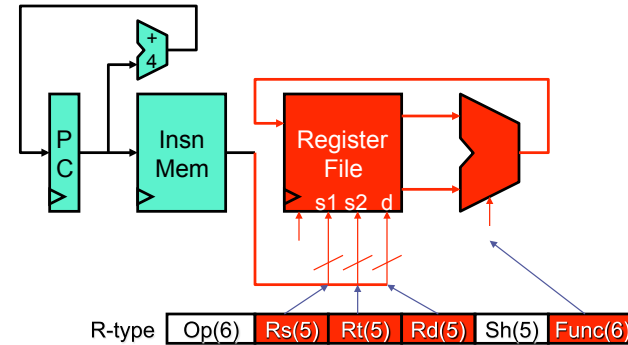
<code>add \$1,\$2,\$3</code>	<code>\$1 = \$2 + \$3</code>	(add)
<code>addi \$1,\$2,3</code>	<code>\$1 = \$2 + 3</code>	(add immed)
<code>lw \$1,4(\$3)</code>	<code>\$1 = Memory[4+\$3]</code>	(load)
<code>sw \$1,4(\$3)</code>	<code>Memory[4+\$3] = \$1</code>	(store)
<code>beq \$1,\$2,PC_relative_target</code>		(branch equal)
<code>j absolute_target</code>		(unconditional jump)
- Why only these?
 - Most other instructions are the same from datapath viewpoint
 - The one's that aren't are left for you to figure out

Start With Fetch



- PC and instruction memory (Harvard architecture, for now)
- A +4 incremter computes default next instruction PC
- How would Verilog for this look given insn memory as interface?

First Instruction: **add**



- Add register file
- Add arithmetic/logical unit (ALU)

Wire Select in Verilog

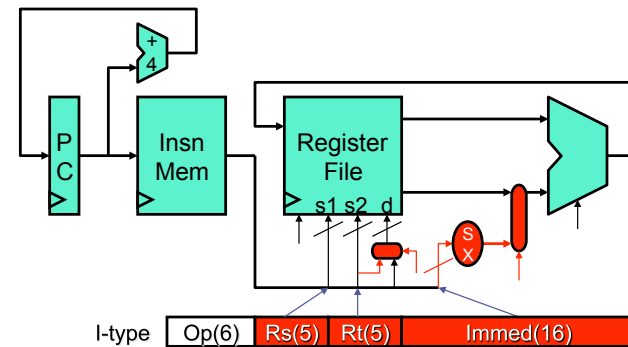
- How to rip out individual fields of an insn? **Wire select**

```

wire [31:0] insn;
wire [5:0] op = insn[31:26];
wire [4:0] rs = insn[25:21];
wire [4:0] rt = insn[20:16];
wire [4:0] rd = insn[15:11];
wire [4:0] sh = insn[10:6];
wire [5:0] func = insn[5:0];
    
```

R-type **Op(6)** **Rs(5)** **Rt(5)** **Rd(5)** **Sh(5)** **Func(6)**

Second Instruction: **addi**



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

Verilog Wire Concatenation

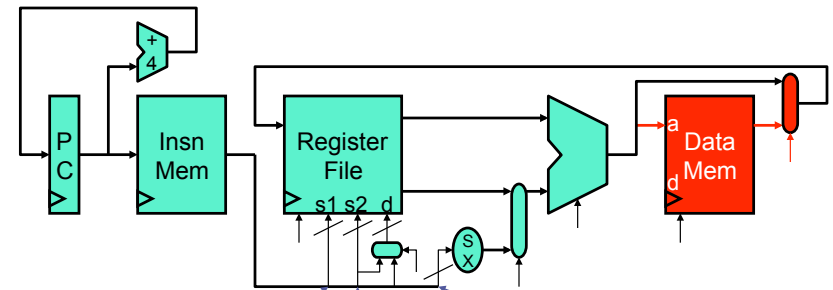
- Recall two Verilog constructs
 - Wire concatenation:** {bus0, bus1, ... , busn}
 - Wire repeat:** {repeat_x_times{w0}}
- How do you specify sign extension? **Wire concatenation**

```

wire [31:0] insn;
wire [15:0] imm16 = insn[15:0];
wire [31:0] sximm16 = {{16{imm16[15]}}, imm16};
    
```

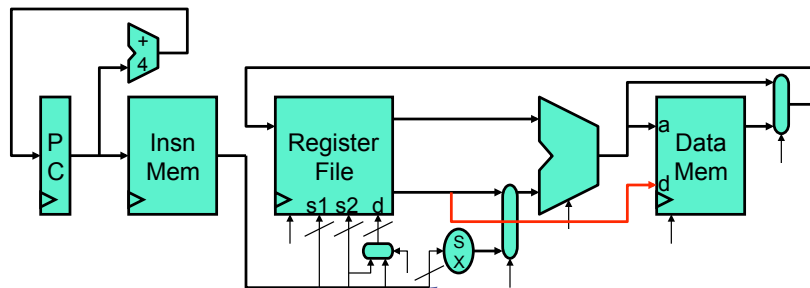


Third Instruction: **lw**



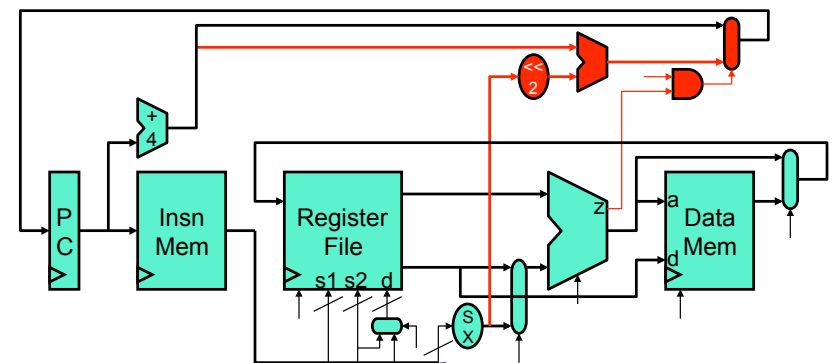
- Add data memory, address is ALU output
- Add register write data mux to select memory output or ALU output

Fourth Instruction: **sw**



- Add path from second input register to data memory data input

Fifth Instruction: **beq**



- Add left shift unit and adder to compute PC-relative branch target
- Add PC input mux to select PC+4 or branch target

Another Use of Wire Concatenation

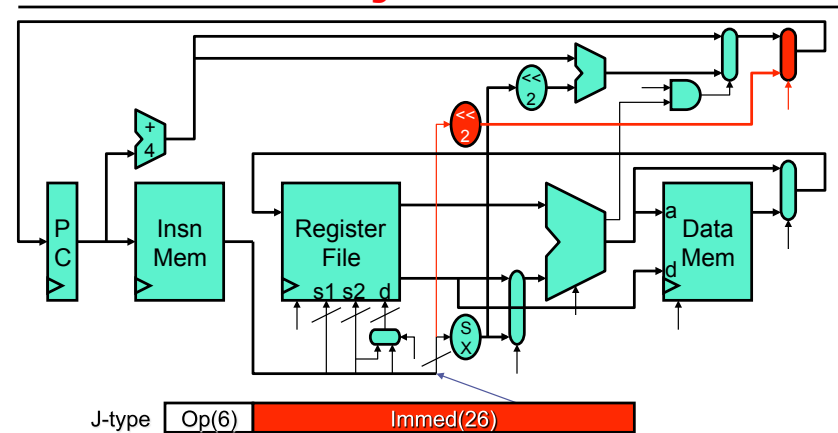
- How do you do $\ll 2$? **Wire concatenation**

```

wire [31:0] insn;
wire [25:0] imm26 = insn[25:0];
wire [31:0] imm26_shifted_by_2 = {4'b0000, imm26, 2'b00};
        
```



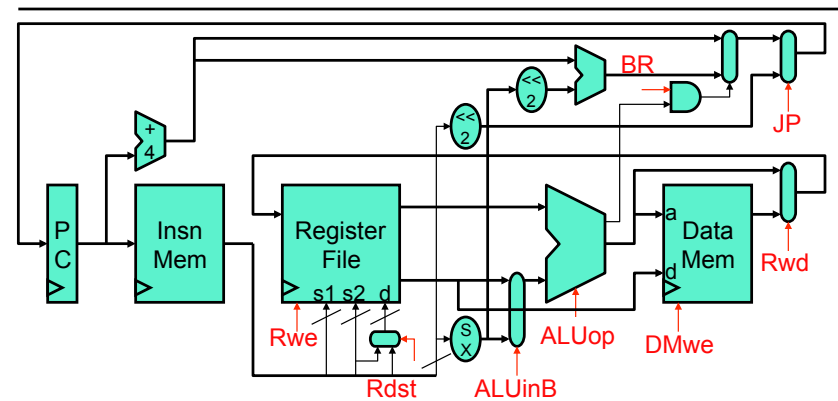
Sixth Instruction: j



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

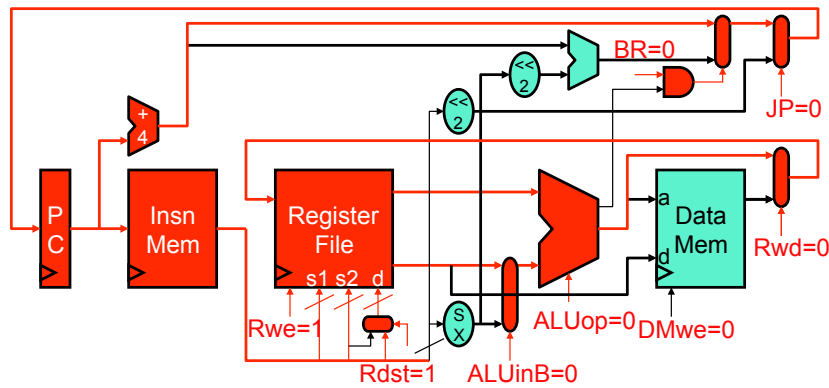
MIPS Control

What Is Control?

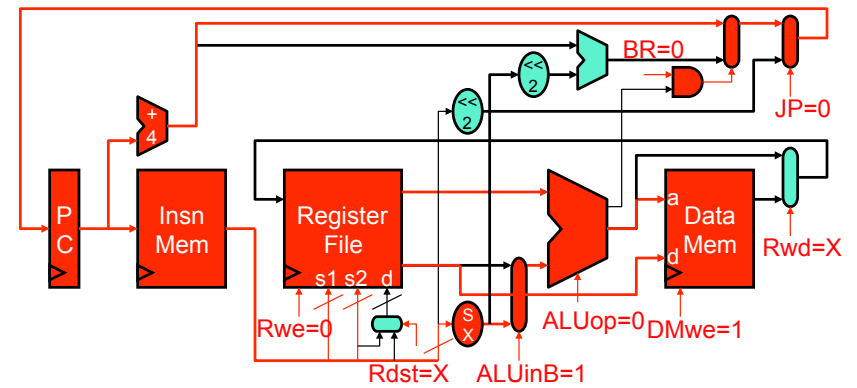


- 9 signals control flow of data through this datapath
 - MUX selectors, or register/memory write enable signals
 - A real datapath has 300-500 control signals

Example: Control for **add**

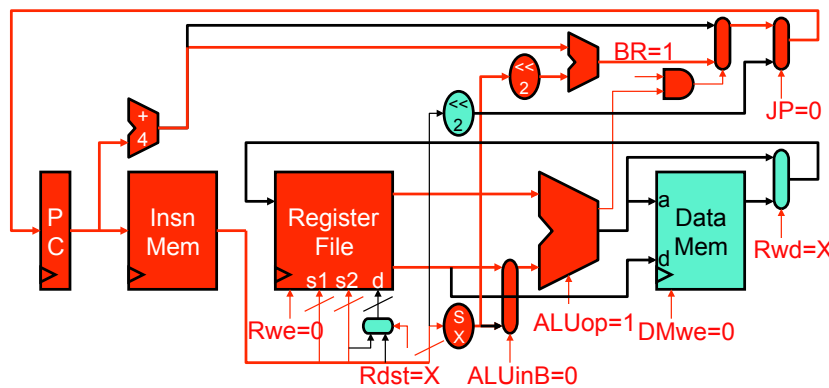


Example: Control for **sw**



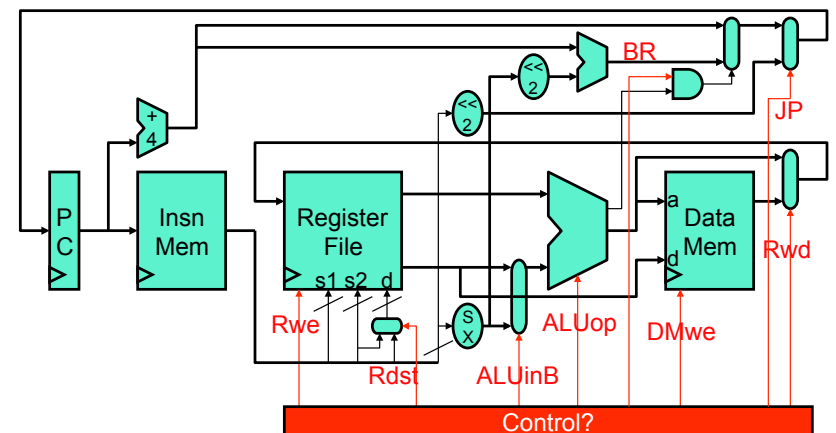
- Difference between **sw** and **add** is 5 signals
 - 3 if you don't count the X (don't care) signals

Example: Control for **beq**



- Difference between **sw** and **beq** is only 4 signals

How Is Control Implemented?

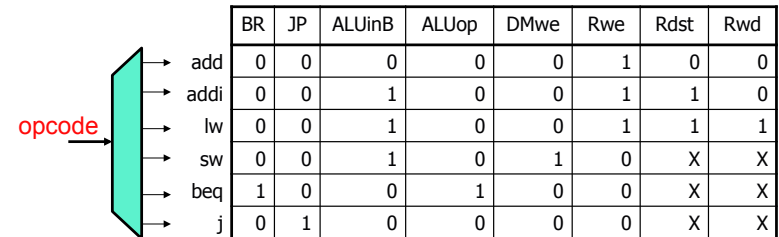


Implementing Control

- Each instruction has a unique set of control signals
 - Most are function of opcode
 - Some may be encoded in the instruction itself
 - E.g., the ALUop signal is some portion of the MIPS Func field
 - + Simplifies controller implementation
 - Requires careful ISA design

Control Implementation: ROM

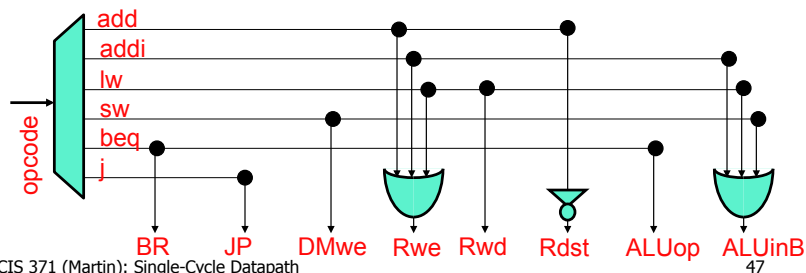
- ROM (read only memory)**: like a RAM but unwritable
 - Bits in data words are control signals
 - Lines indexed by opcode
 - Example: ROM control for 6-insn MIPS datapath
 - X is "don't care"



	BR	JP	ALUinB	ALUop	DMwe	Rwe	Rdst	Rwd
add	0	0	0	0	0	1	0	0
addi	0	0	1	0	0	1	1	0
lw	0	0	1	0	0	1	1	1
sw	0	0	1	0	1	0	X	X
beq	1	0	0	1	0	0	X	X
j	0	1	0	0	0	0	X	X

Control Implementation: Logic

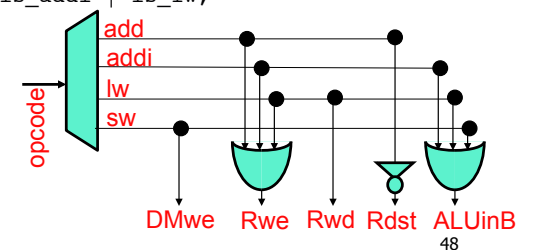
- Real machines have 100+ insns 300+ control signals
 - 30,000+ control bits (~4KB)
 - Not huge, but hard to make faster than datapath (important!)
- Alternative: **logic gates** or "random logic" (unstructured)
 - Exploits the observation: many signals have few 1s or few 0s
 - Example: random logic control for 6-insn MIPS datapath



Control Logic in Verilog

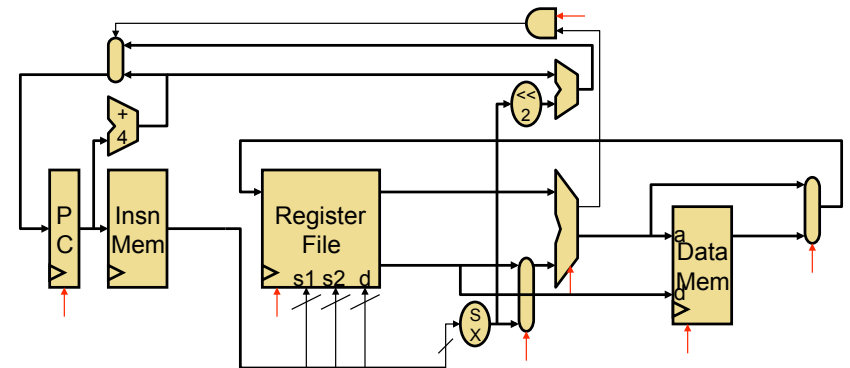
```

wire [31:0] insn;
wire [5:0] func = insn[5:0]
wire [5:0] opcode = insn[31:26];
wire is_add = ((opcode == 6'h00) & (func == 6'h20));
wire is_addi = (opcode == 6'h0F);
wire is_lw = (opcode == 6'h23);
wire is_sw = (opcode == 6'h2A);
wire ALUinB = is_addi | is_lw | is_sw;
wire Rwe = is_add | is_addi | is_lw;
wire Rwd = is_lw;
wire Rdst = ~is_add;
wire DMwe = is_sw;
    
```



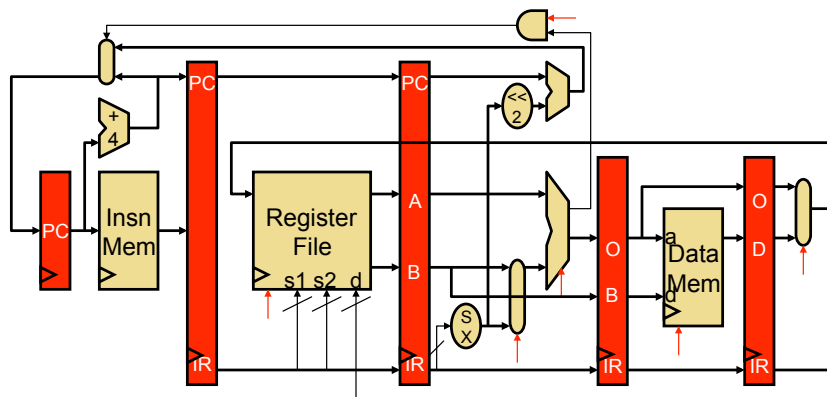
Single-Cycle Performance

Single-Cycle Datapath Performance



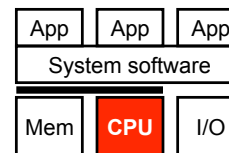
- One cycle per instruction (CPI)
- **Clock cycle time proportional to worst-case logic delay**
 - In this datapath: insn fetch, decode, register read, ALU, data memory access, write register
 - Can we do better?

Foreshadowing: Pipelined Datapath



- Split datapath into multiple stages
 - Assembly line analogy
 - 5 stages results in up to 5x clock & performance improvement

Summary



- Datapath storage elements
- MIPS Datapath
- MIPS Control