MIPS Pipeline (In Verilog)

EE685, Fall 2024

Hank Dietz

http://aggregate.org/hankd/

University of Kentucky

Different Implementations

- **Multi-cycle** MIPS, multiple CPI:

  http://aggregate.org/EE380/multiv.html

- **Single-cycle**, 1 CPI, but slow clock:

  http://aggregate.org/EE380/onebeq.html

- **Pipelined**, multiple CPI, but fast clock and *throughput* up to 1 instruction/cycle

Single-Cycle MIPS Design

- Process one instruction at a time…
- Here's a **multi-cycle** MIPS:

  http://aggregate.org/EE380/multiv.html

  - One instruction, **multiple clock cycles**
  - **Minimal HW**, state machine control
- Our **single-cycle** MIPS design:
  - Each instruction takes **one clock cycle**
  - Lots of hardware, and **not very fast**...
  - **No state machine in control logic**

Single-Cycle MIPS Design

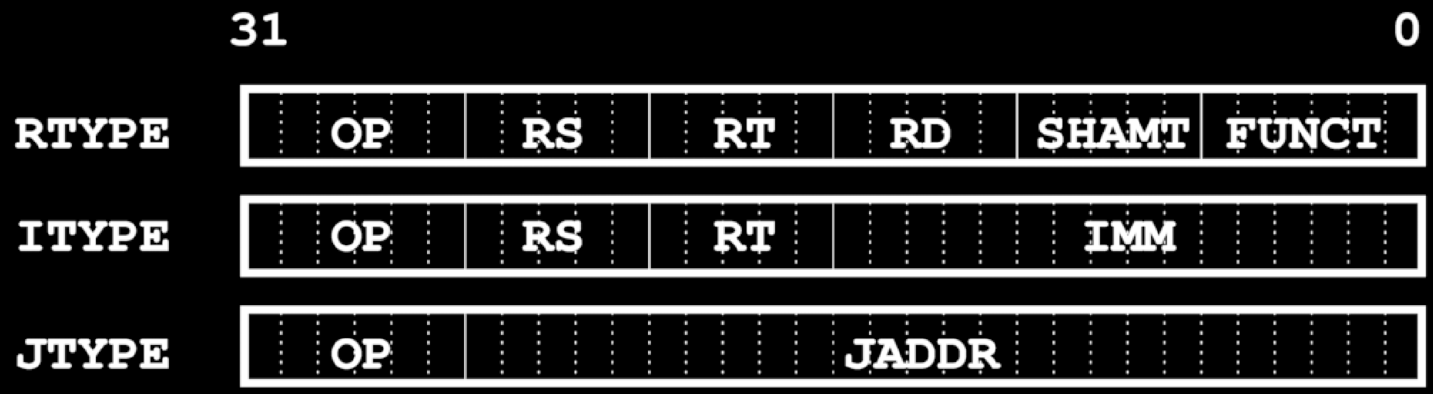
- This is **not** a design you'd really build… it's a step toward a **pipelined design**
- We're not going to design it all at once
  - Can **incrementally design & test**
  - Start by implementing one instruction
  - Handle an instruction sequence
  - Keep adding instructions…
- **Learn the process**, not the design

Types Of Things In MIPS

- Want consistent attributes across all parts of the design, e.g., word size
- Easier to debug and maintain abstracted; e.g., `reg [31:0] t;` holds a word or address?

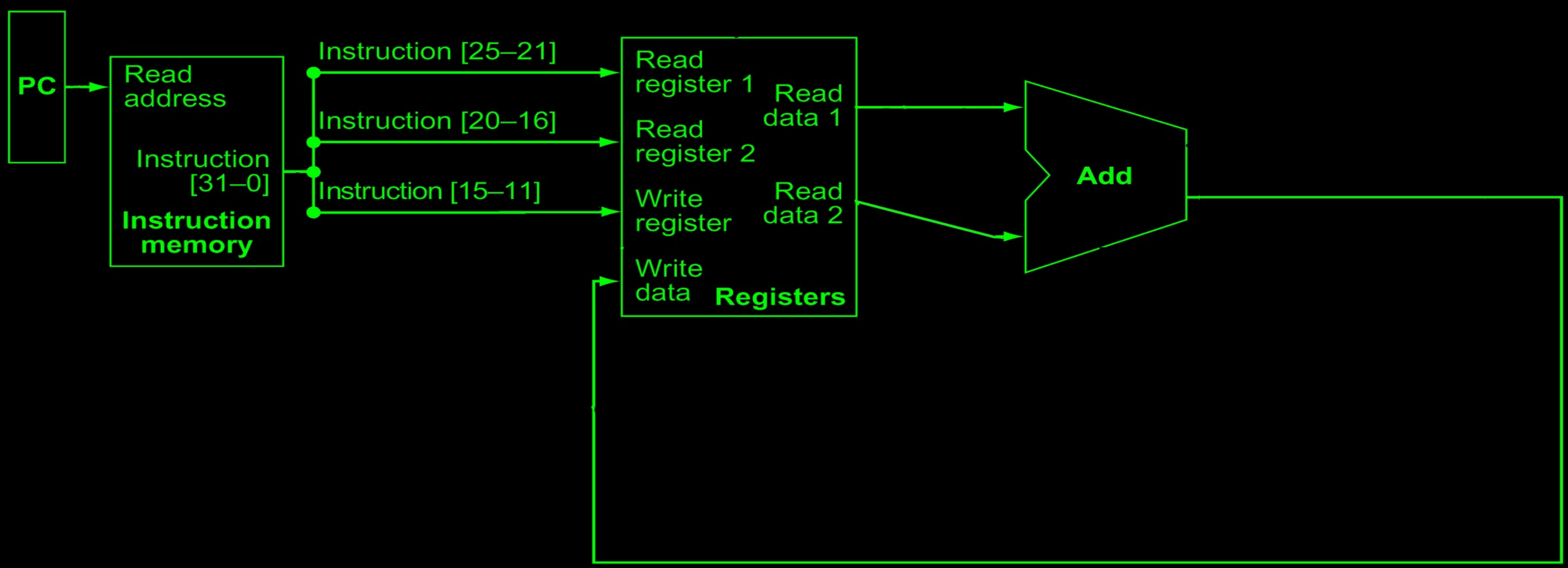
```
// Types
`define WORD    [31:0] // size of a data word
`define ADDR    [31:0] // size of a memory address
`define INST    [31:0] // size of an instruction
`define REG     [4:0]  // size of a register number
`define REGCNT  [31:0] // register count
`define MEMCNT  [511:0]// memory count implemented
`define OPCODE  [5:0]  // 6-bit opcodes
```

MIPS Instruction Fields




```
// Fields
`define OP    [31:26] // opcode field
`define RS    [25:21] // rs field
`define RT    [20:16] // rt field
`define RD    [15:11] // rd field
`define IMM   [15:0]  // immediate/offset field
`define SHAMT [10:6]  // shift amount
`define FUNCT [5:0]   // function code (opcode extension)
`define JADDR [25:0]  // jump address field
```

Let’s start with `addu`



`addu $rd,$rs,$rt`

Now to `addu`

- The **`addu $rd, $rs, $rt`** instruction:
  - Fetch the instruction
  - Read values from registers **`$rs`** and **`$rt`**
  - Add them
  - Write result into **`$rd`**

```
assign ir = m[pc];

always @(posedge clk) begin
    r[ir `RD] <= r[ir `RS] + r[ir `RT];
    halt <= 1;
end
```

The Processor Testbench

```
// Testbench options
`define RUNTIME 100       // How long simulator can run
`define CLKDEL  1         // CLocK transition delay

// Testbench
module bench;
reg reset = 1; reg clk = 0; wire halt;
processor PE(halt, reset, clk);
initial begin
  #`CLKDEL clk = 1;
  #`CLKDEL clk = 0;
  reset = 0;
  while (($time < `RUNTIME) && !halt) begin
    #`CLKDEL clk = 1;
    #`CLKDEL clk = 0;
  end
end
endmodule
```

How do we know it works?

- Need to put an instruction in memory

```
`define JPACK(R,O,J) begin R`OP=0; R`JADDR=J; end
`define IPACK(R,O,S,T,I) begin R`OP=O; R`RS=S; R`RT=T; \
  R`IMM=I; end
`define RPACK(R,S,T,D,SH,FU) begin R`OP=`RTYPE; R`RS=S; \
  R`RT=T; R`RD=D; R`SHAMT=SH; R`FUNCT=FU; end

initial `RPACK(m[0], 2, 3, 1, 0, `ADDU);
```

- Need some way to see what happens

```
`define TRACE  1  // enable simulation trace

`ifdef  TRACE
      $display(… );
`endif
```

Complete MIPS Processor*

that only understands one* **`addu`

- Can run it here:
  `http://aggregate.org/EE380/oneaddu.html`
- Impressed?

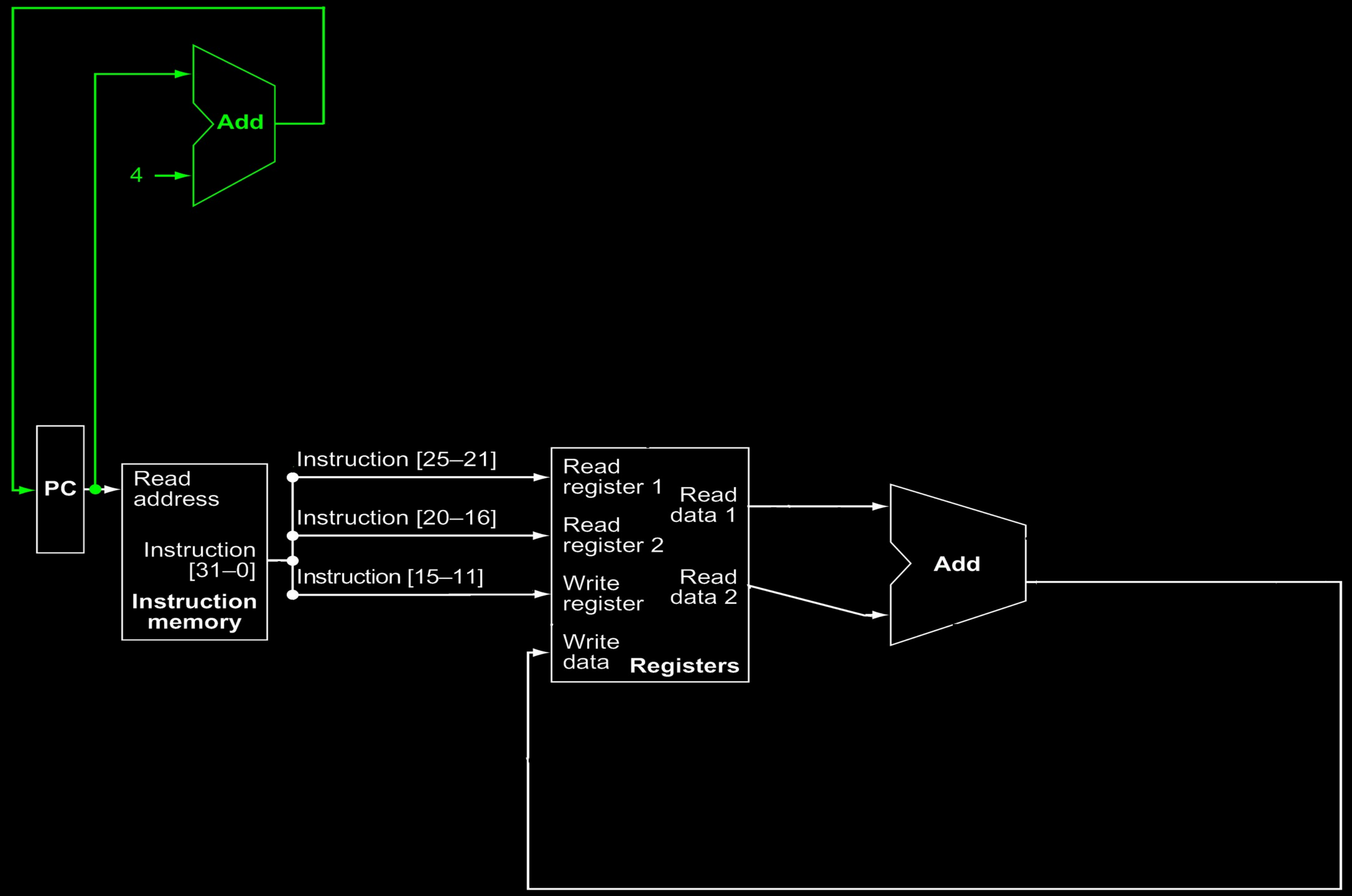
Complete MIPS Processor*

that only understands one* **`addu`

- Can run it here:

  `http://aggregate.org/EE380/oneaddu.html`

- Impressed?
- OK, how about we make it understand how to execute a sequence of **`addu`**?


Add
4
PC
Read
address
Instruction
[31–0]
Instruction
memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
Registers
Add

```
addu $rd,$rs,$rt  ...
```

To execute a sequence of `addu`

- We need to build the PC incrementer:

```
assign PCAdd = (pc + 4);
```

- We also need to check for a legal instruction:

```
if ((ir `OP != `RTYPE) || (ir `FUNCT != `ADDU)) …
```

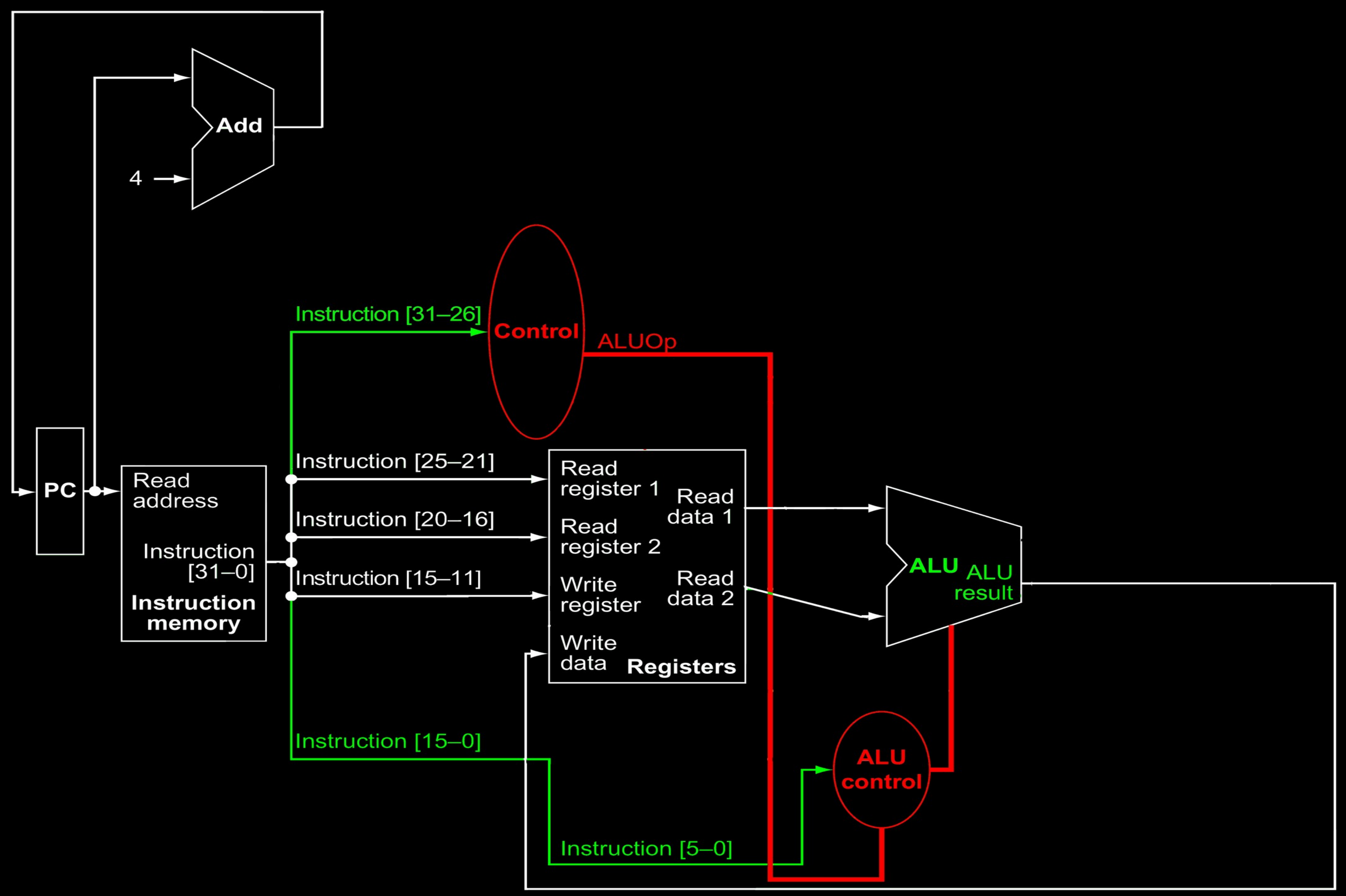
- Can run it here:
  http://aggregate.org/EE380/oneaddus.html

Other RTYPE Instructions?

- There are lots of instructions like **addu**:

```
addu $rd,$rs,$rt        $rd = $rs + $rt
sltu $rd,$rs,$rt        $rd = $rs < $rt
and  $rd,$rs,$rt        $rd = $rs & $rt
or   $rd,$rs,$rt        $rd = $rs | $rt
xor  $rd,$rs,$rt        $rd = $rs ^ $rt
subu $rd,$rs,$rt        $rd = $rs - $rt
```

- Handle them all the same, except in ALU




```
addu $rd,$rs,$rt    subu $rd,$rs,$rt    slt  $rd,$rs,$rt
and  $rd,$rs,$rt    or   $rd,$rs,$rt    xor  $rd,$rs,$rt
```

Decoding RTYPE Inst.

```
// Decode OP, FUNCT into one 7-bit EXTOP
module decode(xop, ir);
output reg `EXTOP xop;  // decoded 7-bit op
input `INST ir;         // instruction

always @(ir) begin
  case (ir `OP)
    `RTYPE: case (ir `FUNCT)
        `ADDU, `SUBU,
        `AND, `OR, `XOR,
        `SLTU:    xop = `F(ir `FUNCT);
        default:  xop = `TRAP; // trap illegal instruction
      endcase
    default:      xop = `TRAP; // trap illegal instruction
  endcase
end
endmodule
```

ALU for RTYPE Inst.

```
// General-purpose ALU
module alu(res, xop, top, bot);
output reg `WORD res;   // combinatorial result
input `EXTOP xop;       // extended operation
input `WORD top, bot;   // top & bottom inputs

// combinatorial always using sensitivity list
// output declared as reg, but never use <=
always @(xop or top or bot) begin
  case (xop)
    `F(`ADDU):  res = (top + bot);
    `F(`SLTU):  res = (top < bot);
    `F(`AND):   res = (top & bot);
    `F(`OR):    res = (top | bot);
    `F(`XOR):   res = (top ^ bot);
    `F(`SUBU):  res = (top - bot);
    // should always cover all possible values
    default:   res = top;
  endcase end endmodule
```

Handle All RTYPE Inst.

- There's a bit of wiring to tie stuff together...

```
// Function unit wiring
wire `ADDR PCAdd;
wire `EXTOP ALUcontrol;
wire `WORD ALUresult;

// Control logic
assign ALUOp = (ir `OP);

// Function units
decode DECODE(ALUcontrol, ir);
alu    ALU(ALUresult, ALUcontrol, r[ir `RS], r[ir `RT]);
```

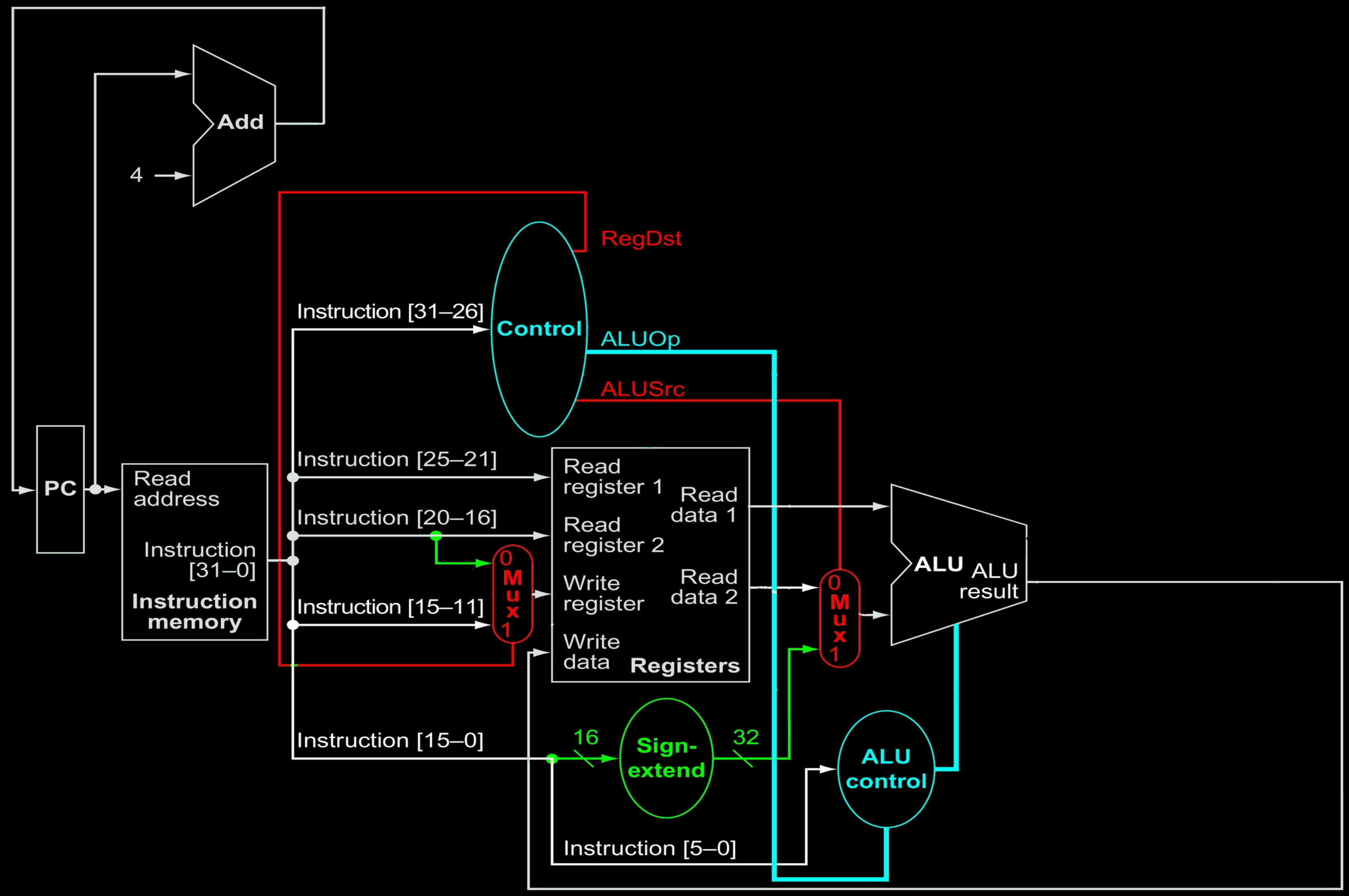
- Can run it here:
  http://aggregate.org/EE380/onertypes.html

How About Immediates?

- Immediates use the ALU similarly…

`addiu $rt,$rs,imm`	$rt = $rs + imm
`sltiu $rt,$rs,imm`	$rt = $rs < imm
`andi $rt,$rs,imm`	$rt = $rs & imm
`ori $rt,$rs,imm`	$rt = $rs | imm
`xori $rt,$rs,imm`	$rt = $rs ^ imm
`lui $rt,imm`	$rt = imm << 16

- Lui is odd, but is encoded as $0 for $rs




```
addiu $rt,$rs,imm    sltiu $rt,$rs,imm    lui   $rt,imm
andi  $rt,$rs,imm    ori   $rt,$rs,imm    xori  $rt,$rs,imm
```

Immediate Data Paths

- Need immediate sign extender for 16 to 32 bits:

```
wire `WORD Signextend;

assign Signextend = {{16{ir[15]}}, ir `IMM};
```

- Need muxes to select ALU inputs, reg to write:

```
wire ALUSrc;
wire `REG RegDstMux;
wire `WORD ALUSrcMux;

assign RegDst = (ALUOp == `RTYPE);
assign RegDstMux = (RegDst ? ir `RD : ir `RT);
assign ALUSrcMux = (ALUSrc ? Signextend : r[ir `RT]);
```

Decoding Imm Inst.

```
// Decode OP, FUNCT into one 7-bit EXTOP
module decode(xop, ir);
output reg `EXTOP xop;  // decoded 7-bit op
input `INST ir;         // instruction

always @(ir) begin
  case (ir `OP)
    `RTYPE: case (ir `FUNCT)
        `ADDU, `SUBU,
        `AND, `OR, `XOR,
        `SLTU:    xop = `F(ir `FUNCT);
        default:  xop = `TRAP; // trap illegal instruction
      endcase
    `ADDIU, `SLTIU,
    `ANDI, `ORI, `XORI,
    `LUI:         xop = ir `OP;
    default:      xop = `TRAP; // trap illegal instruction
  endcase end endmodule
```

ALU for Imm Instructions

```
// General-purpose ALU
module alu(res, xop, top, bot);
output reg `WORD res;   // combinatorial result
input `EXTOP xop;       // extended operation
input `WORD top, bot;   // top & bottom inputs

// combinatorial always using sensitivity list
// output declared as reg, but never use <=
always @(xop or top or bot) begin
  case (xop)
    `ADDIU, `F(`ADDU):  res = (top + bot);
    `SLTIU, `F(`SLTU):  res = (top < bot);
    `ANDI, `F(`AND):    res = (top & bot);
    `ORI, `F(`OR):      res = (top | bot);
    `XORI, `F(`XOR):    res = (top ^ bot);
    `LUI:               res = (bot << 16);
    `F(`SUBU):          res = (top - bot);
    // should always cover all possible values
    default:   res = top;
  endcase end endmodule
```

RTYPE & Immediate...

- Of course, we add new test cases:

```
`IPACK(m[6], `ADDIU, 3, 1, -1);
`IPACK(m[7], `SLTIU, 5, 1, 12345);
`IPACK(m[8], `ANDI, 3, 1, 3);
`IPACK(m[9], `ORI, 3, 1, 3);
`IPACK(m[10], `XORI, 3, 1, 3);
`IPACK(m[11], `LUI, 0, 1, 1);
```

- The TRACE output had to be upgraded:

```
if (ir `OP) $display("%d: OP=%x RS=%d RT=%d IMM=%x",
pc, ir `OP, ir `RS, ir `RT, ir `IMM); else
$display("%d: OP=%x RS=%d RT=%d RD=%d SHAMT=%d FUNCT=%x",
pc, ir `OP, ir `RS, ir `RT, ir `RD, ir `SHAMT, ir `FUNCT);
```

- Can run it here:
  http://aggregate.org/EE380/oneimms.html

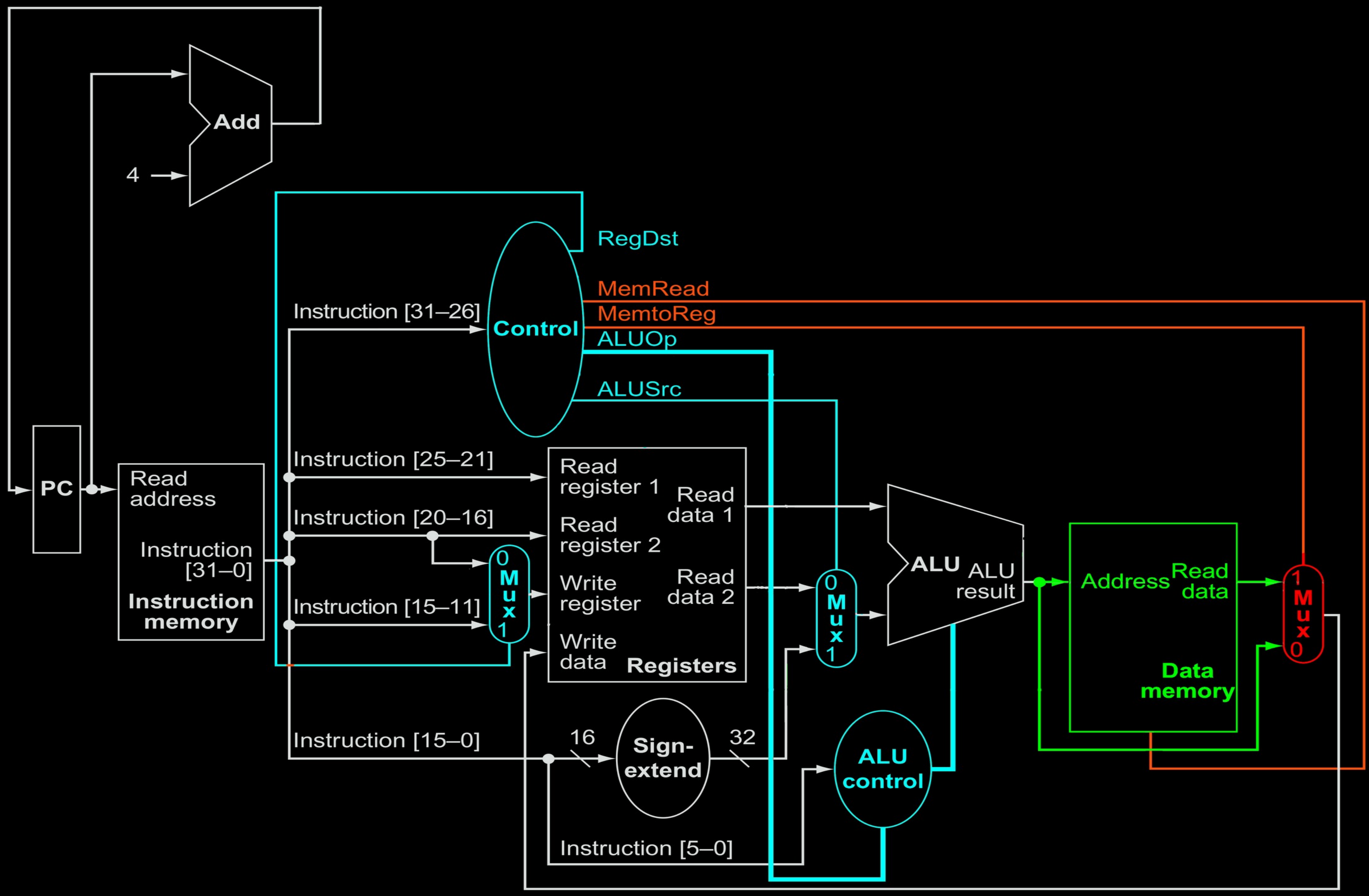
Load From Memory

- There's only one load word instruction:

  `lw $rt,imm($rs)`

  $rt = memory[imm + $rs]

- It's sort-of an **`addiu`**, but uses the ALU result as the memory address to read


Add
4
RegDst
MemRead
MemtoReg
ALUOp
ALUSrc
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
0 Mux 1
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
0 Mux 1
ALU
ALU result
Address
Read data
Data memory
1 Mux 0
Instruction [15–0]
16
Sign-extend
32
ALU control
Instruction [5–0]

```
lw    $rt,imm($rs)
```

So, What Does `lw` Need?

- Need MemtoReg mux:

```
MemtoReg = (ir `OP == `LW);
assign MemtoRegMux = (MemtoReg ? m[ALUresult >> 2] :
                                 ALUresult);
```

- Of course, we add a new test case:

```
`IPACK(m[12], `LW, 2, 1, 255)
m[256] = 22;
```

- Can run it here:
  http://aggregate.org/EE380/onelw.html

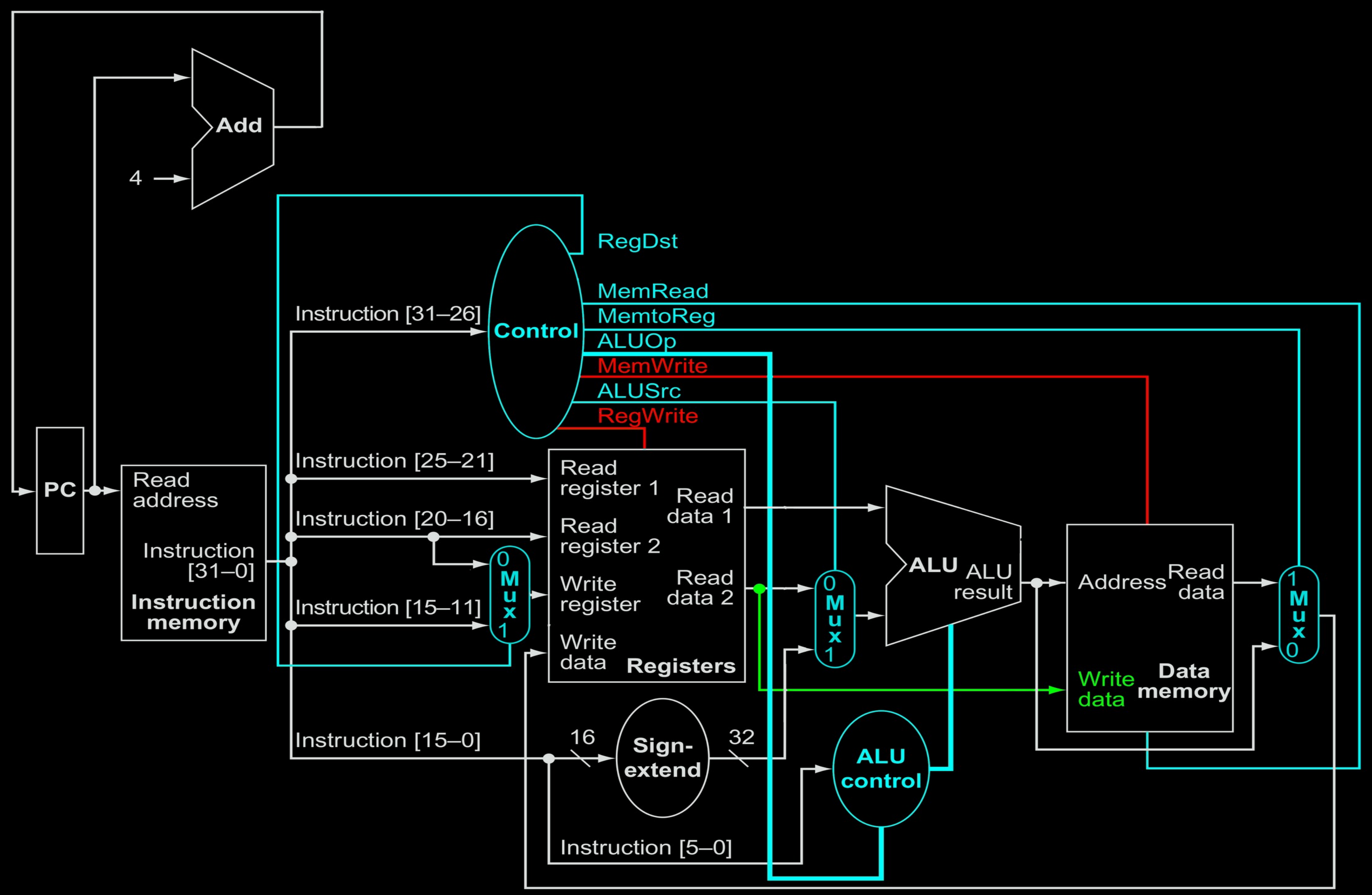
Store To Memory

- There's only one store word instruction:

  ```
  sw $rt,imm($rs)
  ```

  memory[imm + $rs] = $rt

- It's sort-of an `lw`…



Add
4
RegDst
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
PC
Read address
Instruction [31–0]
Instruction memory
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
0 Mux 1
Sign-extend
16
32
ALU
ALU result
ALU control
Address
Read data
Write data
Data memory
1 Mux 0


```
sw    $rt,imm($rs)
```

What Changes For **`sw`**?

- Very similar to **`lw`**, but:
  - Need to route data to memory
  - Unlike every other instruction thus far, sw doesn't write to a register

```
if (MemWrite) m[ALUresult >> 2] <= r[ir `RT];
if (RegWrite) r[RegDstMux] <= MemtoRegMux;
```

- Of course, we also add a new test case:

```
`IPACK(m[12], `SW, 0, 2, 255)
```

- Can run it here:
  http://aggregate.org/EE380/onesw.html

Don’t We Need Control Flow?

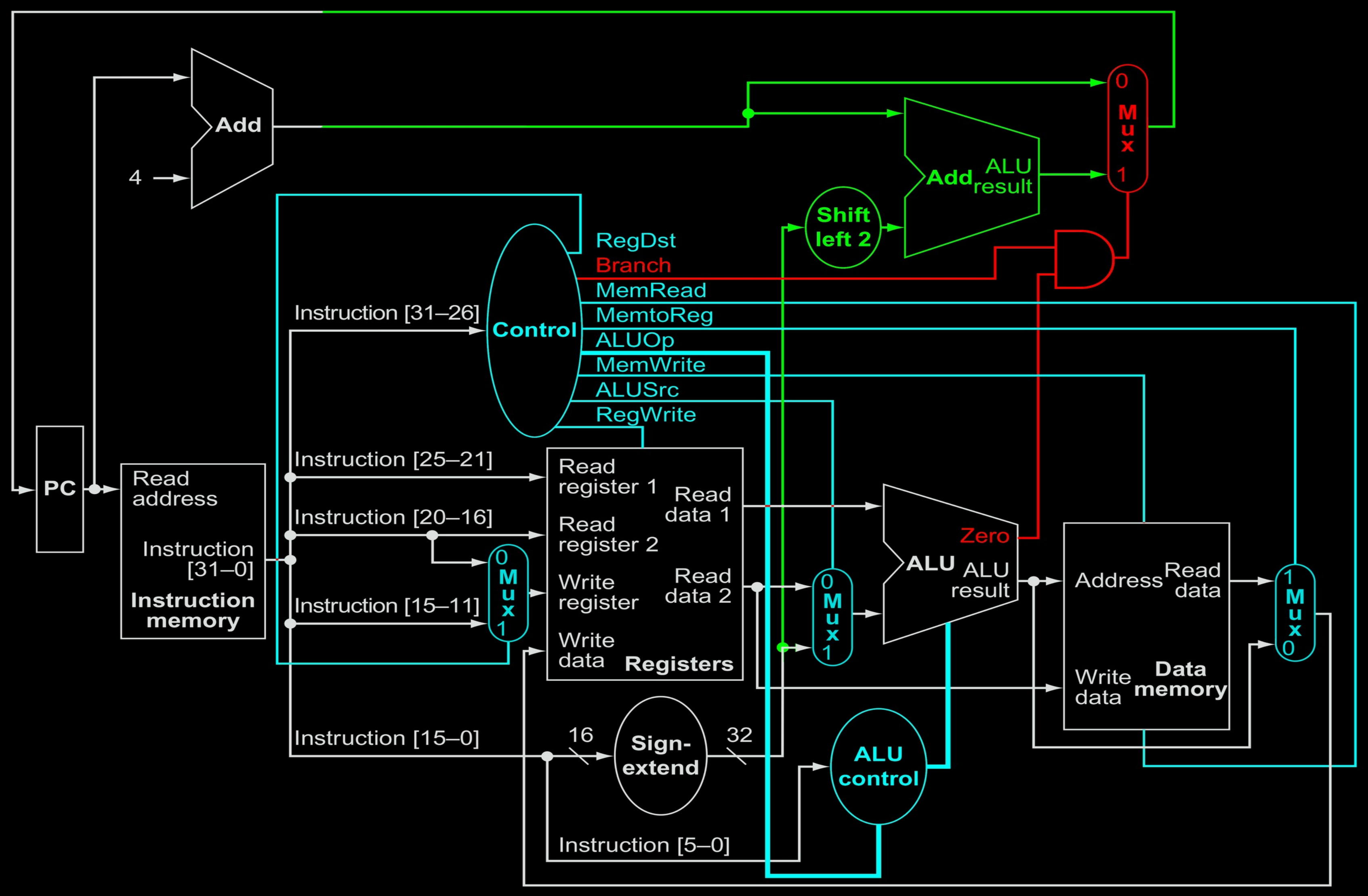
- How about a branch equals?

  `beq $rs,$rt,lab`

  if ($rs == $rt) pc = (pc + 4) + (offset * 4)

  where offset = (lab – (pc + 4)) / 4

- Of course, offset is really imm… and we shift by 2 rather than multiply by 4




```
beq   $rt,$rs,lab   # offset = (lab - (PC + 4)) >> 2
```

What Changes For `beq`?

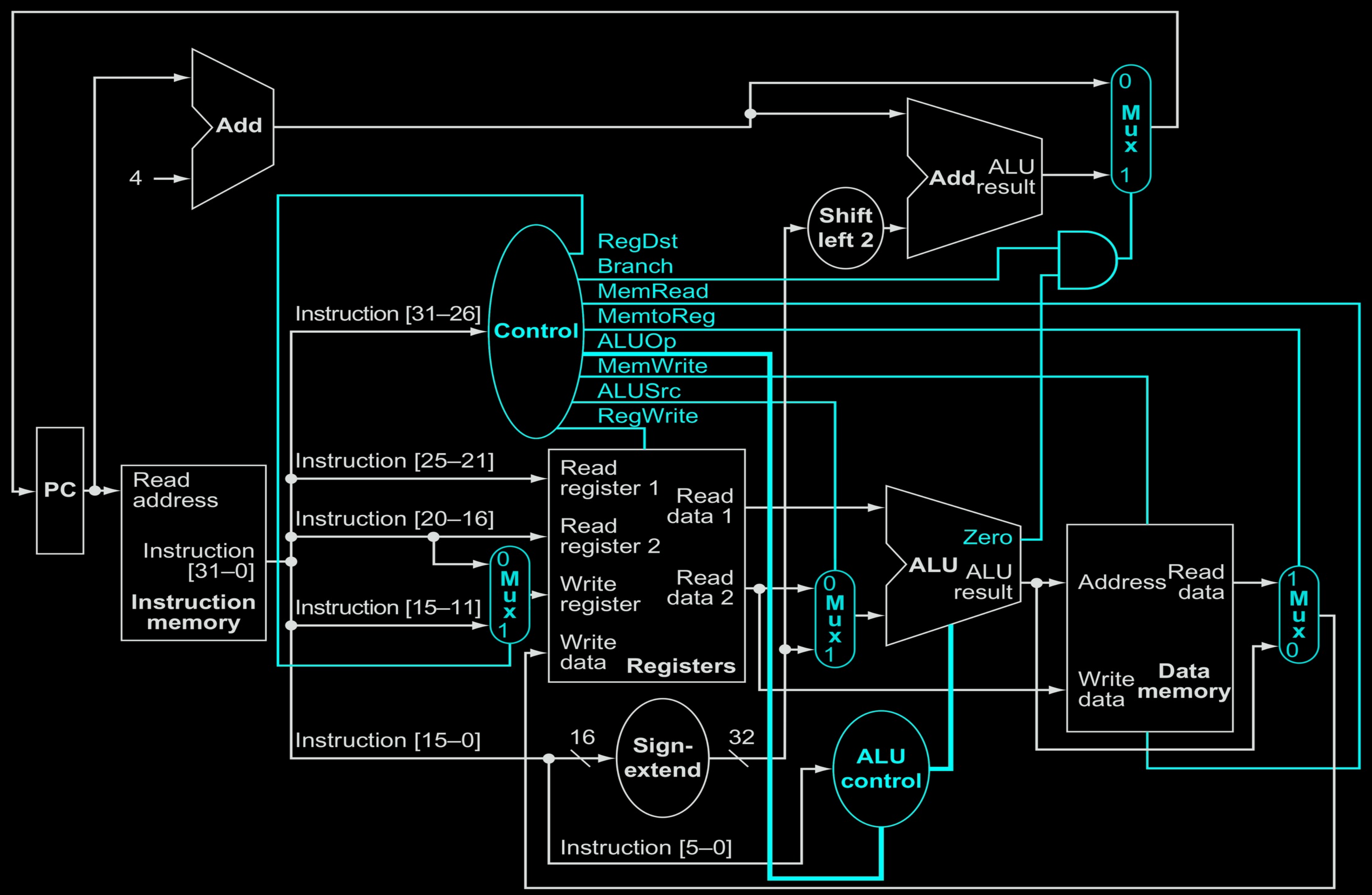
- Need a shift-by-2 unit and another adder

```
assign Shiftleft2 = (Signextend << 2);
assign BranchAdd = (PCAdd + Shiftleft2);
```

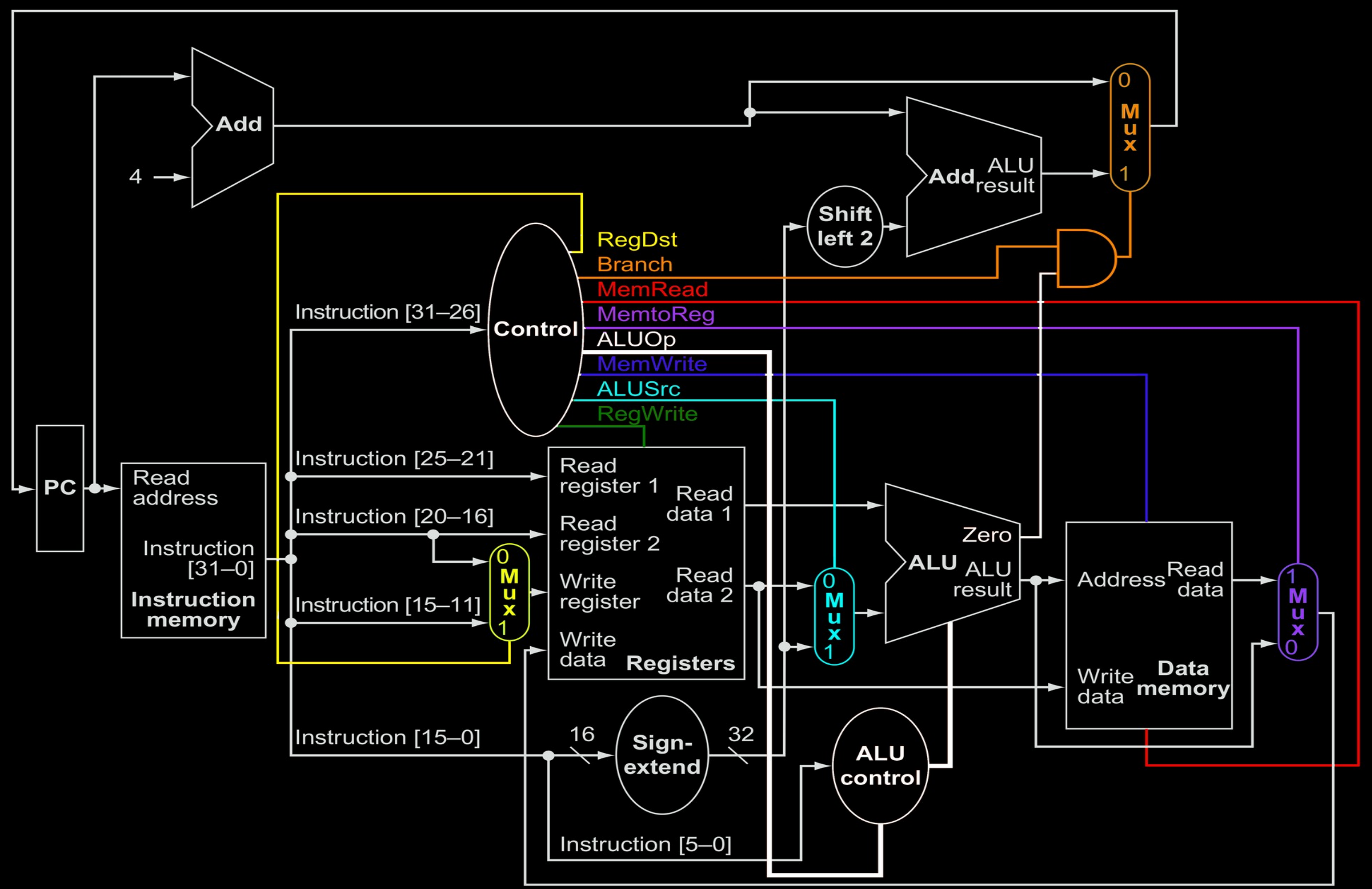
- Need ALU to have a zero flag (use $rs - $rt) and mux to use it

```
assign ALUSrc = ((ir `OP != `RTYPE) && (ir `OP != `BEQ));
assign zero = (res == 0);
assign Branch = (ir `OP == `BEQ);
assign BranchZeroMux = ((Branch & Zero) ? BranchAdd:PCAdd);
```

- Can run it here:
  http://aggregate.org/EE380/onebeq.html



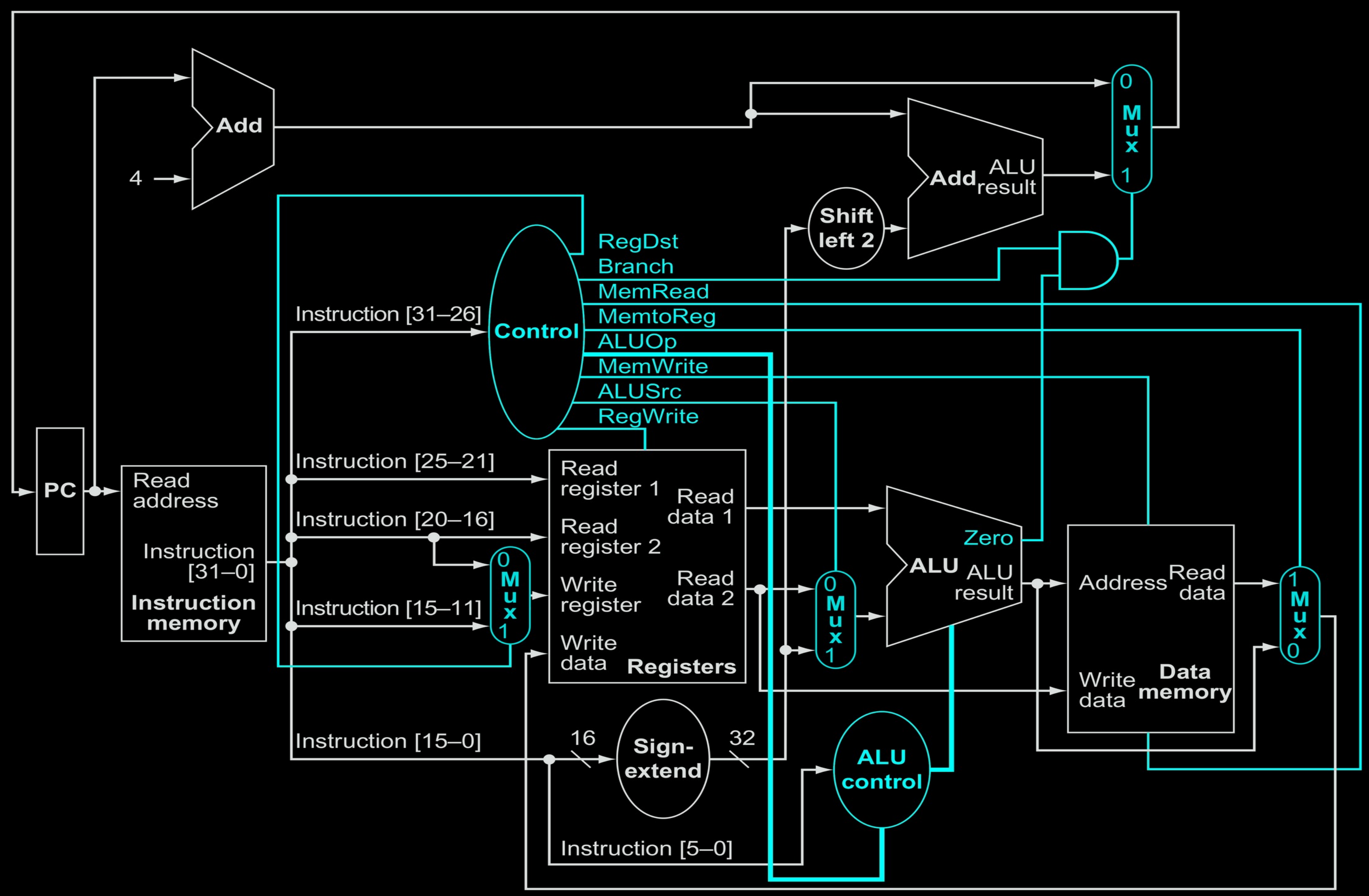

```
# can incrementally add paths for more instructions
# always try to reuse as much hardware as possible
```



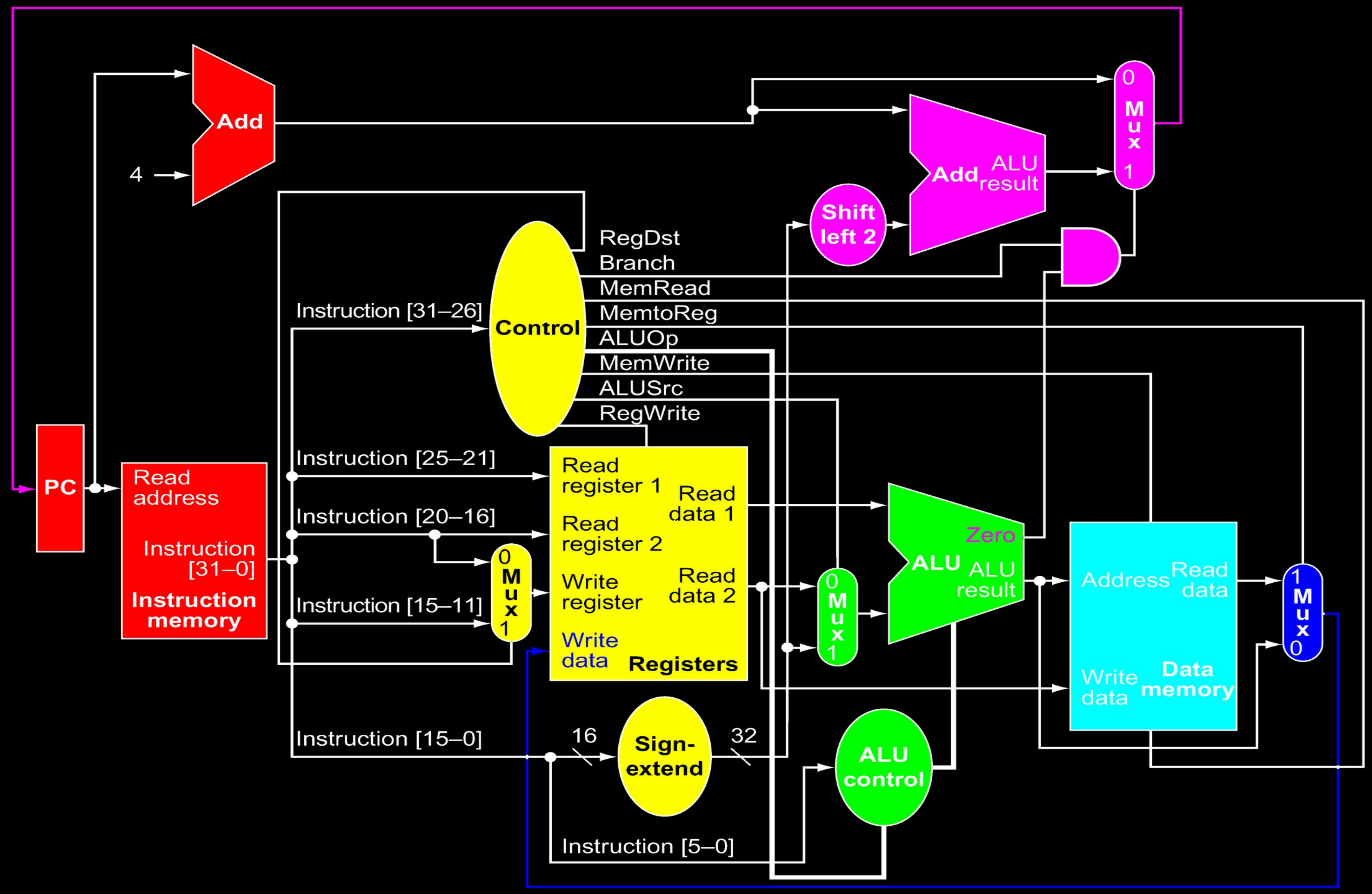
	RegDst	Branch	MemRead	MemtoReg	MemWrite	ALUSrc	RegWrite
addu	1	0	0	0	0	0	1
addiu	0	0	0	0	0	1	1
lw	0	0	1	1	0	1	1
sw	X	0	0	X	1	1	0
beq	X	1	0	X	0	0	0

Basic Pipelining

- Single-cycle **control signals move through the pipe** along with the data
- Divide single-cycle into **equal-delay stages**, adding **buffers between**
  - Ideally, ***n*** stages gives ***n*** X throughput
  - Usually **<** ***n*** X, and ***n*** can't be huge
- Throughput comes from having useful work in all stages – avoiding **bubbles**

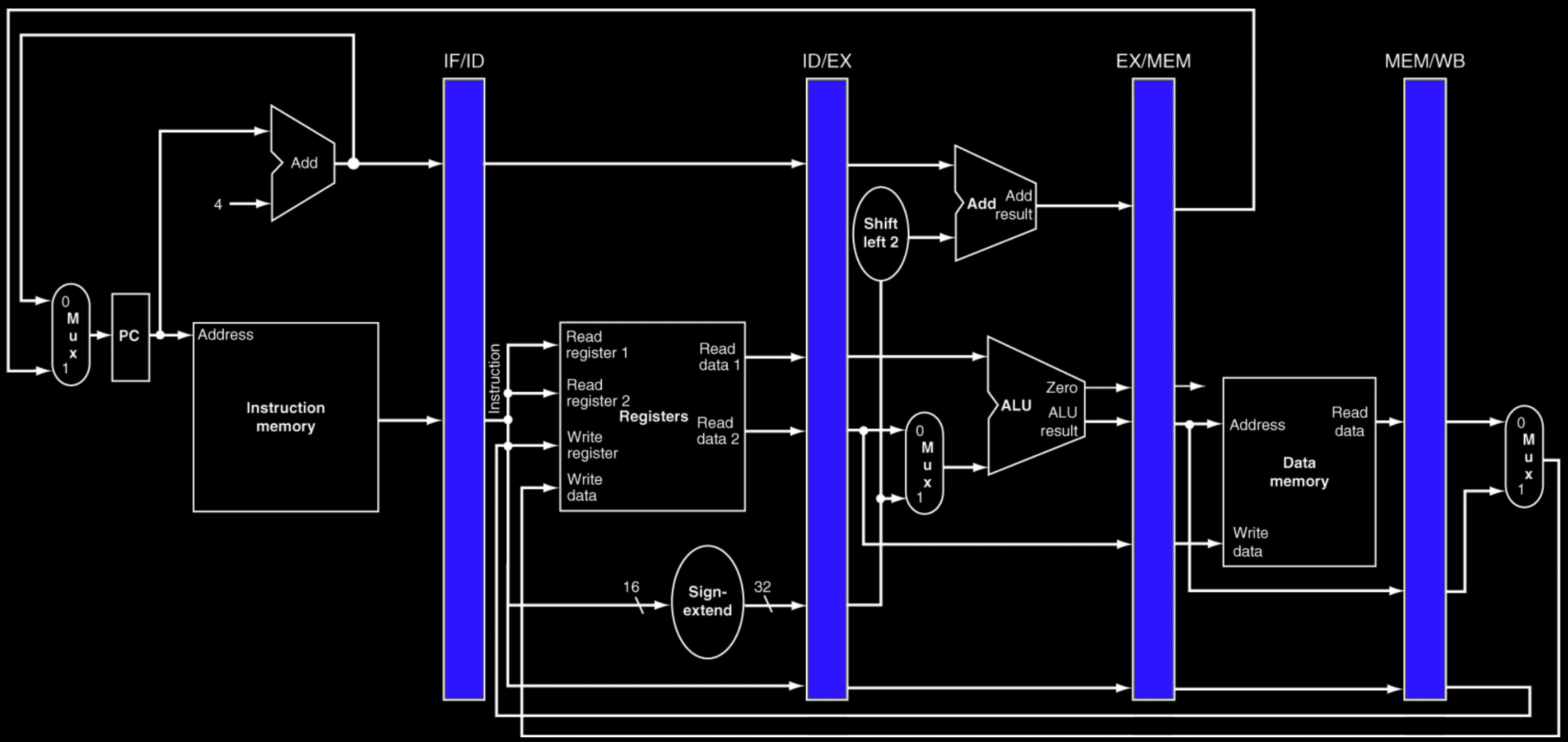


The single cycle design...



IF: Instruction Fetch EX: Execute WB: Write Back

ID: Instruction Decode MEM: Memory Access what is this?



- Add buffers between pipe stages...

Pipeline Throughput

IF	ID	EX	MEM	WB
addu $t0,$t1,$t2				–
	addu $t0,$t1,$t2			–
		addu $t0,$t1,$t2		–
			addu $t0,$t1,$t2	–
				addu $t0,$t1,$t2
and $t3,$t4,$t5				–
	and $t3,$t4,$t5			–
		and $t3,$t4,$t5		–
			and $t3,$t4,$t5	–
				and $t3,$t4,$t5

. . .

IF	ID	EX	MEM	WB
addu $t0,$t1,$t2				–
and $t3,$t4,$t5	addu $t0,$t1,$t2			–
addiu $t6,$0,1	and $t3,$t4,$t5	addu $t0,$t1,$t2		–
lw $t7,0($t8)	addiu $t6,$0,1	and $t3,$t4,$t5	addu $t0,$t1,$t2	–
sw $t1,4($t9)	lw $t7,0($t8)	addiu $t6,$0,1	and $t3,$t4,$t5	addu $t0,$t1,$t2
beq $t1,$t2,lab	sw $t1,4($t9)	lw $t7,0($t8)	addiu $t6,$0,1	and $t3,$t4,$t5
…	beq $t1,$t2,lab	sw $t1,4($t9)	lw $t7,0($t8)	addiu $t6,$0,1
…	…	beq $t1,$t2,lab	sw $t1,4($t9)	lw $t7,0($t8)
…	…	…	beq $t1,$t2,lab	sw $t1,4($t9)
…	…	…	…	beq $t1,$t2,lab

Pipeline Bubble: nop

IF	ID	EX	MEM	WB
addu $t0,$t1,$t2				–
and $t3,$t4,$t5	addu $t0,$t1,$t2			–
addiu $t6,$0,1	and $t3,$t4,$t5	addu $t0,$t1,$t2		–
lw $t7,0($t8)	addiu $t6,$0,1	and $t3,$t4,$t5	addu $t0,$t1,$t2	–
sw $t1,4($t9)	lw $t7,0($t8)	addiu $t6,$0,1	and $t3,$t4,$t5	addu $t0,$t1,$t2
beq $t1,$t2,lab	sw $t1,4($t9)	lw $t7,0($t8)	addiu $t6,$0,1	and $t3,$t4,$t5
…	beq $t1,$t2,lab	sw $t1,4($t9)	lw $t7,0($t8)	addiu $t6,$0,1
…	…	beq $t1,$t2,lab	sw $t1,4($t9)	lw $t7,0($t8)
…	…	…	beq $t1,$t2,lab	sw $t1,4($t9)
…	…	…	…	beq $t1,$t2,lab

IF	ID	EX	MEM	WB
addu $t0,$t1,$t2				–
and $t3,$t4,$t5	addu $t0,$t1,$t2			–
addiu $t6,$0,1	and $t3,$t4,$t5	addu $t0,$t1,$t2		–
lw $t7,0($t8)	addiu $t6,$0,1	and $t3,$t4,$t5	addu $t0,$t1,$t2	–
sw $t1,4($t7)	lw $t7,0($t8)	addiu $t6,$0,1	and $t3,$t4,$t5	addu $t0,$t1,$t2
beq $t1,$t2,lab	sw $t1,4($t7)	lw $t7,0($t8)	addiu $t6,$0,1	and $t3,$t4,$t5
beq $t1,$t2,lab	sw $t1,4($t7)	nop	lw $t7,0($t8)	addiu $t6,$0,1
beq $t1,$t2,lab	sw $t1,4($t7)	nop	nop	lw $t7,0($t8)
beq $t1,$t2,lab	sw $t1,4($t7)	nop	nop	nop
…	beq $t1,$t2,lab	sw $t1,4($t7)	nop	nop
…	…	beq $t1,$t2,lab	sw $t1,4($t7)	nop
…	…	…	beq $t1,$t2,lab	sw $t1,4($t7)
…	…	…	…	beq $t1,$t2,lab

Setting The Stages

Instruction	IF	ID	EX	MEM	WB	Circuit delay
addu	250	100	**300**	–	100	750ps
and	250	100	100	–	100	550ps
addiu	250	100	**300**	–	100	750ps
lw	**250**	**100**	**300**	**250**	**100**	**1000ps**
sw	250	100	**300**	100	100	850ps
beq	250	100	**300**	–	–	750ps

- Single-cycle limited by `lw` to **1GHz clock**
- 5-stage pipeline limited by `EX`
  - 300ps latency allows **3.33GHz** clock
  - If added buffers take 100ps, **2.5GHz**
- Multi-cycle might be **5 cycles @2.5GHz**

Why A Bubble?

- Structural hazards: can't do 2 things on one unit of HW – single-cycle cures this!
- Data dependence: need results from previous computations – not yet done?
- Control dependence
  - Computation of branch target address
  - Conditional branch/jump taken or not?

Dependence Analysis

- Use or R: reads the value of a name
- Def or W: binds a new value to a name
- True dep.: carries a value, D→U, RAW
  `add $t0,$t1,$t2    or $t3,$t0,$t4`
- Anti-dep.: kills a value, U←D, WAR
  `add $t0,$t1,$t2    or $t1,$t3,$t4`
- Output dep.: kills a value, D→D, WAW
  `add $t0,$t1,$t2    or $t0,$t3,$t4`

When Dependence Matters

- True dependence causes a delay

```
add $t0,$t1,$t2
or  $t3,$t0,$t4  //wait for $t0
```

- **Other types don't**

How To Handle Dependence

- Programmer/assembler pads with NOPs
  - Violates ISA concept (how many NOPs?) too little padding gives wrong answers, too much wastes time
  - Makes program bigger

IF	ID	EX	MEM	WB
add $t0,$t1,$t2	…	…	…	…
nop	add $t0,$t1,$t2	…	…	…
nop	nop	add $t0,$t1,$t2	…	…
nop	nop	nop	add $t0,$t1,$t2	…
or $t3,$t0,$t4	nop	nop	nop	add $t0,$t1,$t2
…	or $t3,$t0,$t4	nop	nop	nop
…	…	or $t3,$t0,$t4	nop	nop
…	…	…	or $t3,$t0,$t4	nop
…	…	…	…	or $t3,$t0,$t4

How To Handle Dependence

- Hardware interlock
  - rs or rt in ID is dest in EX, MEM, WB
  - Detects dep. & stalls until satisfied
  - **`nop`** is **`or`** with side-effects disabled: MemRead, MemWrite, RegWrite, & Branch

IF	ID	EX	MEM	WB
`add $t0,$t1,$t2`	…	…	…	…
`or $t3,$t0,$t4`	`add $t0,$t1,$t2`	…	…	…
…	`or $t3,$t0,$t4`	`add $t0,$t1,$t2`	…	…
…	`or $t3,$t0,$t4`	`nop`	`add $t0,$t1,$t2`	…
…	`or $t3,$t0,$t4`	`nop`	`nop`	`add $t0,$t1,$t2`
…	`or $t3,$t0,$t4`	`nop`	`nop`	`nop`
…	…	`or $t3,$t0,$t4`	`nop`	`nop`
…	…	…	`or $t3,$t0,$t4`	`nop`
…	…	…	…	`or $t3,$t0,$t4`

How To Handle Dependence

- Value forwarding: use interlock circuitry to find value & forward it to ID stage out
  - ALU ready from EX, so **NO DELAY**!
  - `lw` ready from MEM, so 1 cycle delay
  - Value still gets stored in register in WB
  - Works great, but adds datapaths…

IF	ID	EX	MEM	WB
`lw $t0,0($t1)`	…	…	…	…
`or $t3,$t0,$t4`	`lw $t0,0($t1)`	…	…	…
…	`or $t3,$t0,$t4`	`lw $t0,0($t1)`	…	…
…	`or $t3,$t0,$t4`	`nop`	`lw $t0,0($t1)`	…
…	…	`or $t3,$t0,$t4`	`nop`	`lw $t0,0($t1)`
…	…	…	`or $t3,$t0,$t4`	`nop`
…	…	…	…	`or $t3,$t0,$t4`

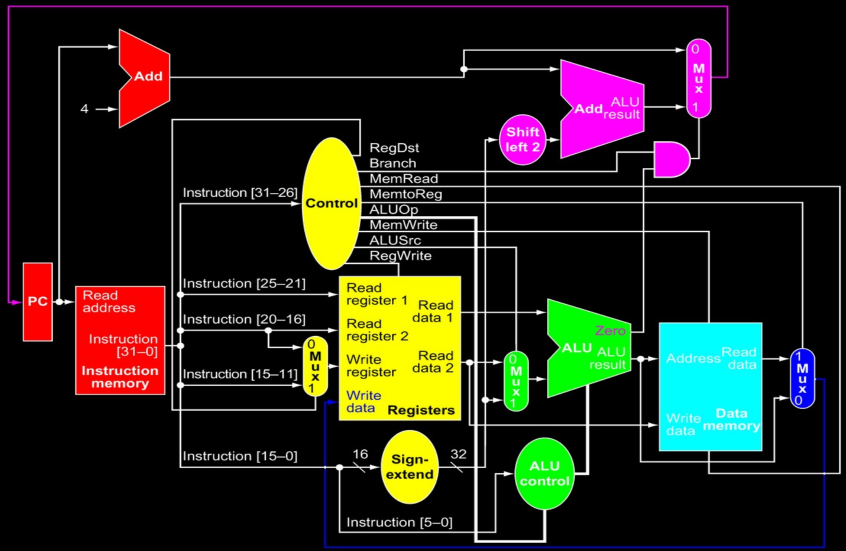
Computing The Branch Target

- Branch instructions encode offset, not address
  - Add PC + offset *typically* takes a cycle
  - Hardware interlock & stall waiting

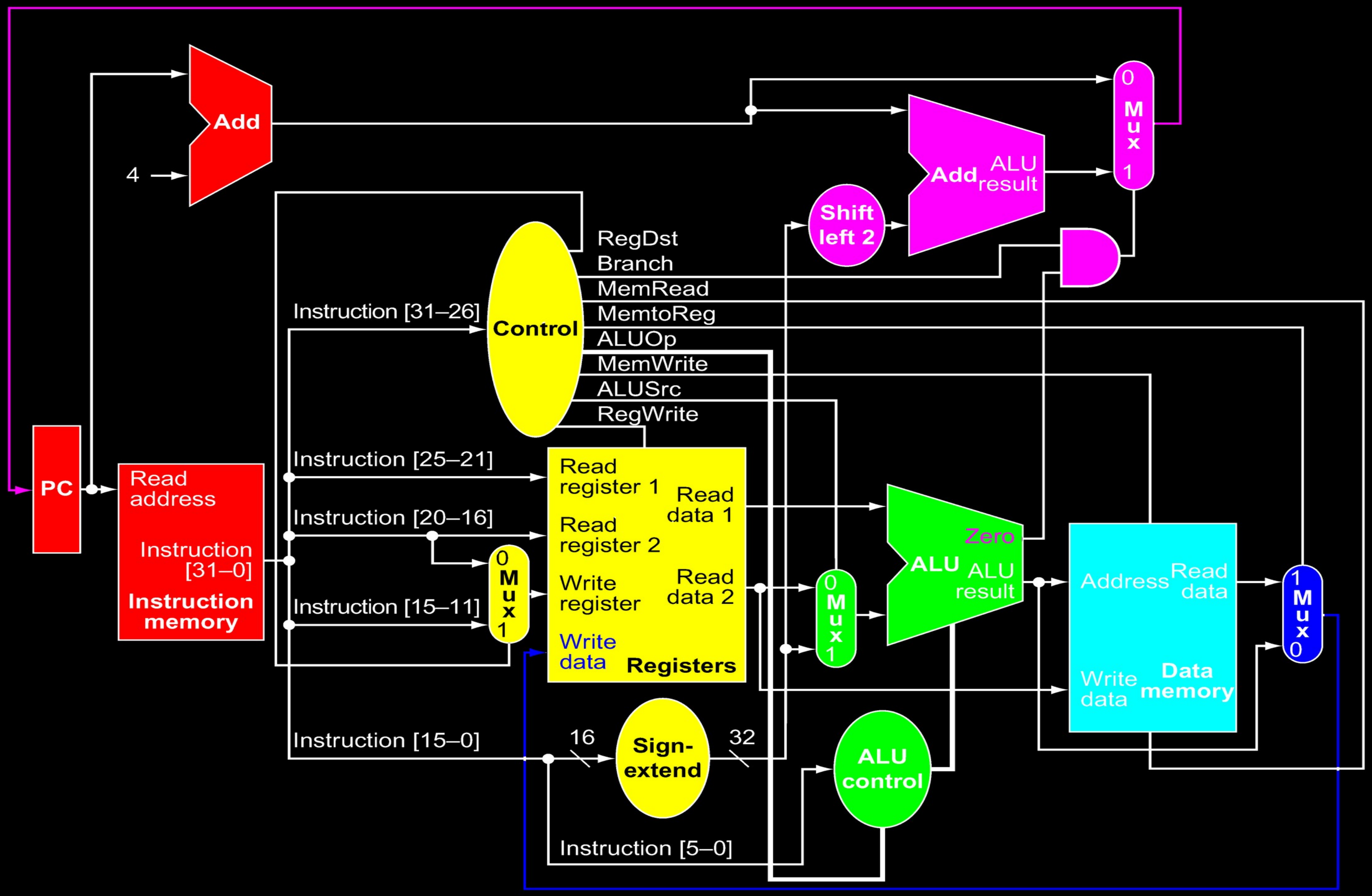
To Take, Or Not To Take?

- When do we know if conditional is taken or not taken?

Determined only after EX stage?

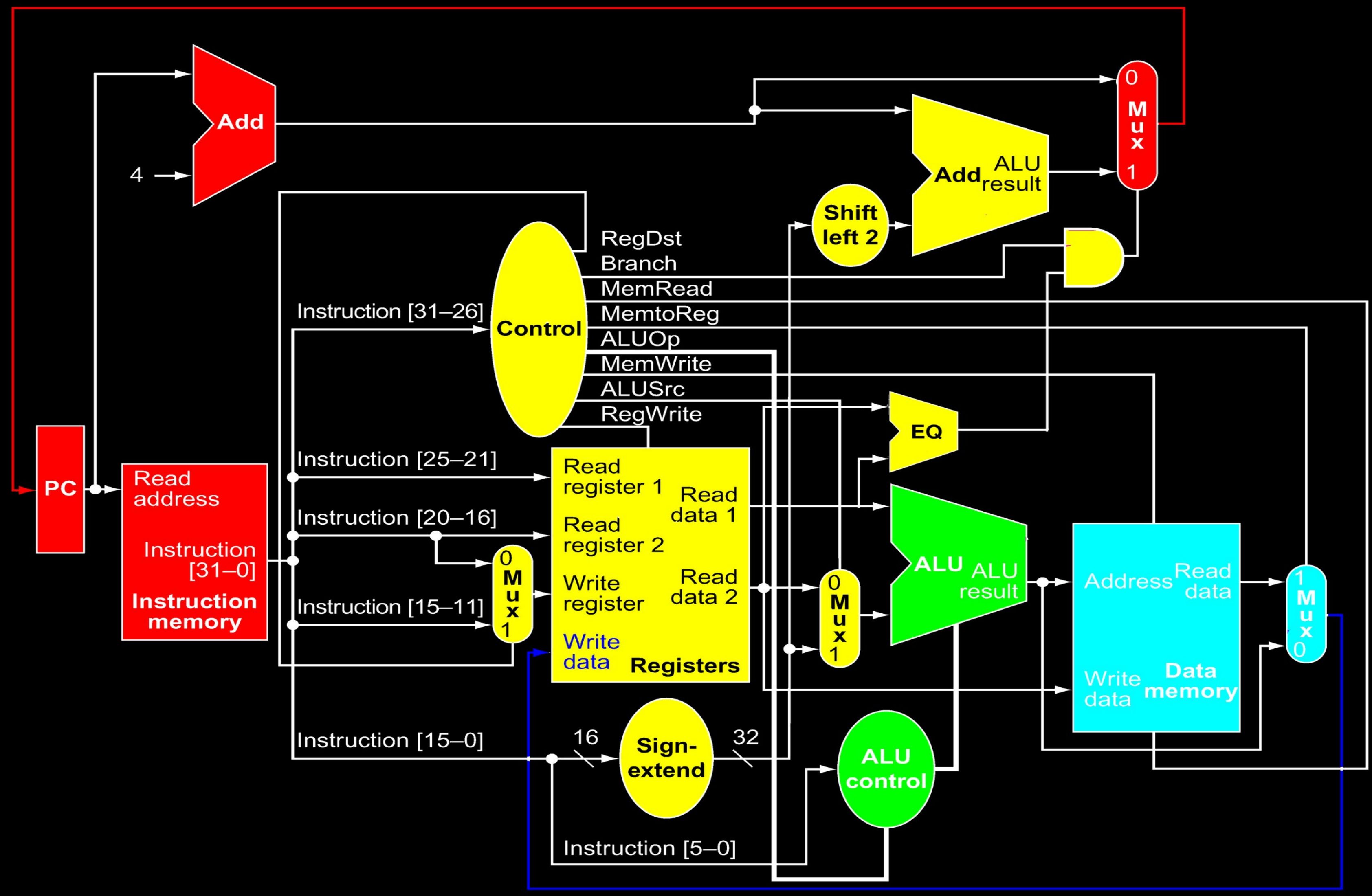


- HW interlock very inefficient!
  - Usually determined in a late pipe stage
  - Blocks ALL progress



IF: Instruction Fetch   EX: Execute   WB: Write Back

ID: Instruction Decode   MEM: Memory Access   what is this?



IF: Instruction Fetch EX: Execute WB: Write Back
ID: Instruction Decode MEM: Memory Access

Branch Prediction.
Do I Feel Lucky?

Guess wrong and some instructions are gonna die

*(well, we actually say they're **squashed**)*

- **Always not taken** – the easiest guess
- Later, we'll discuss better ways to guess…

Verilog Implementation

- Like you'd expect:
  - Can reuse basic single-cycle design
  - Each stage becomes it's own `always`
  - Need multiple copies of some signals, one for each stage that uses them

- Not like you'd expect:
  - Some things don't follow pipe flow
  - Some non-stages should be separate things

Owner Computes

- For example, who updates the PC?
  - IF sets PC = PC + 4
  - ID sets PC = branch target
  - EX, MEM, or WB forces PC = PC because data dependence blocks the pipe
- How to coordinate access to shared data?
  - Multiple readers is fine, with one writer
  - Could use locks, semaphores, etc.
  - Let only one entity update each: **owner** picks value and is only writer

Pipelined Verilog Version

- Organized as parallel-executing chunks:
  - IF stage: reads and writes `IF_`
  - ID stage: reads `IF_`, writes `ID_`
  - EX stage: reads `ID_`, writes `EX_`
  - MEM stage: reads `EX_`, writes `MEM_`
  - WB stage: reads `MEM_`
  - Are we `running`?
  - Are mispredicted inst. `squashed`?
  - Are we `blocked` by data dependence or forwarding values?
  - Simulation tracing support

Pipelined Verilog Version

```
// Testbench options
`define RUNTIME 200     // How long simulator can run
`define CLKDEL  1       // CLocK transition delay
`define TRACE   1       // enable simulation trace

// Types
`define WORD    [31:0]  // size of a data word
`define ADDR    [31:0]  // size of a memory address
`define INST    [31:0]  // size of an instruction
`define REG     [4:0]   // size of a register number
`define REGCNT  [31:0]  // register count
`define MEMCNT  [511:0] // memory count implemented
`define OPCODE  [5:0]   // 6-bit opcodes
`define ALUOP   [3:0]   // Simplified ALU codes

// Fields
`define OP      [31:26] // opcode field
`define RS      [25:21] // rs field
`define RT      [20:16] // rt field
`define RD      [15:11] // rd field
`define IMM     [15:0]  // immediate/offset field
`define SHAMT   [10:6]  // shift ammount
`define FUNCT   [5:0]   // function code (opcode extension)
`define JADDR   [25:0]  // jump address field
`define JPACK(R,O,J)            begin R`OP=O; R`JADDR=J; end
`define IPACK(R,O,S,T,I)        begin R`OP=O; R`RS=S; R`RT=T; R`IMM=I; end
`define RPACK(R,S,T,D,SH,FU)    begin R`OP=`RTYPE; R`RS=S; R`RT=T; R`RD=D; R`SHAMT=SH; R`FUNCT=FU; end
`define NOP     `OR     // Null OPeration is or $0,$0,$0

// Instruction encoding
`define RTYPE   6'h00   // OP field for all RTYPE instructions
`define BEQ     6'h04   // OP field
`define ADDIU   6'h09   // OP field
`define SLTIU   6'h0b   // OP field
`define ANDI    6'h0c   // OP field
`define ORI     6'h0d   // OP field
`define XORI    6'h0e   // OP field
`define LUI     6'h0f   // OP field
`define LW      6'h23   // OP field
`define SW      6'h2b   // OP field
`define ADDU    6'h21   // FUNCT field
`define SUBU    6'h23   // FUNCT field
`define AND     6'h24   // FUNCT field
`define OR      6'h25   // FUNCT field
`define XOR     6'h26   // FUNCT field
`define SLTU    6'h2b   // FUNCT field

// Simplified ALU codes, default to lui
`define ALUAND  4'b0000
`define ALUOR   4'b0001
`define ALUADD  4'b0010
`define ALUSUB  4'b0110
`define ALUSLT  4'b0111
`define ALULUI  4'b1000
`define ALUXOR  4'b1111

// Generic multi-cycle processor
module processor(halt, reset, clk);
output reg halt;
input reset, clk;
reg `WORD m `MEMCNT;
reg `WORD r `REGCNT;

// Initialize register file and memory
initial begin
    r[1] = 22; r[2] = 1; r[3] = 42;
    r[4] = 601; r[5] = 11811; r[6] = -1;
    `RPACK(m[0], 2, 3, 1, 0, `ADDU)
    `RPACK(m[1], 2, 3, 1, 0, `SLTU)
    `RPACK(m[2], 3, 5, 1, 0, `AND)
    `RPACK(m[3], 5, 3, 1, 0, `OR)
    `RPACK(m[4], 3, 5, 1, 0, `XOR)
    `RPACK(m[5], 4, 3, 1, 0, `SUBU)
    `IPACK(m[6], `ADDIU, 3, 1, -1)
    `IPACK(m[7], `SLTIU, 5, 1, 12345)
    `IPACK(m[8], `ANDI, 3, 1, 3)
    `IPACK(m[9], `ORI, 3, 1, 3)
    `IPACK(m[10], `XORI, 3, 1, 3)
    `IPACK(m[11], `LUI, 0, 1, 1)
    `IPACK(m[12], `LW, 2, 1, 1023)
    `IPACK(m[13], `SW, 0, 2, 1024)
    `IPACK(m[14], `ADDIU, 6, 6, 1)
    `IPACK(m[15], `BEQ, 0, 6, -2)
    m[16] = 0;
    m[256] = 22;
end

// IF registers
reg `ADDR IF pc;
reg `INST IF ir;

// ID registers
reg RegDst, Branch, MemRead, MemWrite, ALUSrc, RegWrite, Bad, ID MemRead, ID MemWrite, ID Bad;
reg `ALUOP ALUOp, ID ALUOp;
reg `WORD s, t, imm, ID s, ID t, ID src;
reg `REG ID rd;
reg `ADDR target;
reg squash;

// EX registers
reg EX MemRead, EX MemWrite, EX Bad;
reg `WORD alu, EX alu, EX t;
reg `REG EX rd;

// MEM registers
reg MEM Bad;
reg `WORD v, MEM v;
reg `REG MEM rd;
```

```
// Running state?
wire running;
assign running = ((!halt) && (!reset));

// Squash instruction fetched on a mispredicted branch
wire `INST squashed;
assign squashed = (squash ? `NOP : IF ir);

// Are we blocked by a dependence on rs or rt?
// Also handles all forwarding
wire EX deps, EX dept, MEM deps, MEM dept, WB deps, WB dept, canfwds, canfwdt, dep, blocked;
wire `WORD fwds, fwdt;
assign ID needs = (IF ir `RS != 0);
assign ID needt = (IF ir `RT != 0);
assign EX deps = (IF ir `RS == ID rd);
assign EX dept = (IF ir `RT == ID rd);
assign MEM deps = (IF ir `RS == EX rd);
assign MEM dept = (IF ir `RT == EX rd);
assign WB deps = (IF ir `RS == MEM rd);
assign WB dept = (IF ir `RT == MEM rd);
assign canfwds = ID needs && ((EX deps && !MemRead) || ((!EX deps) && (MEM deps || WB deps)));
assign canfwdt = ID needt && ((EX dept && !MemRead) || ((!EX dept) && (MEM dept || WB dept)));
assign fwds = (EX deps ? alu : (MEM deps ? v : MEM v));
assign fwdt = (EX dept ? alu : (MEM dept ? v : MEM v));
assign dep = ((ID needs && (EX deps || MEM deps || WB deps)) ||
              (ID needt && (EX dept || MEM dept || WB dept)));
assign blocked = (dep && ((ID needs && !canfwds) || (ID needt && !canfwdt)));

// IF: Instruction Fetch stage
always @(posedge clk) if (running && !blocked) begin
  IF ir <= m[(squash ? target : IF pc) >> 2];
  IF pc <= (squash ? target : IF pc) + 4;
end

// ID: Instruction Decode stage
always @(posedge clk) if (running && !ID Bad) begin
  if (blocked) begin
    ID s <= 0;
    ID t <= 0;
    ID src <= 0;
    ID rd <= 0; // not writing
    ID MemRead <= 0; // not reading
    ID ALUOp <= 0;
    ID MemWrite <= 0; // not storing
    ID Bad <= 0;
  end else begin
    // Brute force decode of instruction
    case (squashed `OP)
      `RTYPE: begin
        case (squashed `FUNCT)
          `ADDU:   begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUADD; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `SUBU:   begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUSUB; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `AND:    begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUAND; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `OR:     begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUOR; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `XOR:    begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUXOR; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `SLTU:   begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUSLT; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          default: begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUOR; MemWrite=0; ALUSrc=0; RegWrite=0; Bad=1; end
        endcase
      end
      `BEQ:    begin RegDst=0; Branch=1; MemRead=0; ALUOp=`ALUSUB; MemWrite=0; ALUSrc=0; RegWrite=0; Bad=0; end
      `ADDIU:  begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUADD; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `SLTIU:  begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUSLT; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `ANDI:   begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUAND; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `ORI:    begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUOR; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `XORI:   begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUXOR; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `LUI:    begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALULUI; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `LW:     begin RegDst=0; Branch=0; MemRead=1; ALUOp=`ALUADD; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `SW:     begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUADD; MemWrite=1; ALUSrc=1; RegWrite=0; Bad=0; end
      default: begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUOR; MemWrite=0; ALUSrc=0; RegWrite=0; Bad=1; end
    endcase

    s = (canfwds ? fwds : r[squashed `RS]);
    t = (canfwdt ? fwdt : r[squashed `RT]);
    imm = {{16{squashed[15]}}, squashed `IMM};
    target <= IF pc + {imm[29:0], 2'b00};

    squash <= (Branch && (s == t));
    ID s <= s;
    ID t <= t;
    ID src <= (ALUSrc ? imm : t);
    ID rd <= (RegWrite ? (RegDst ? squashed `RD : squashed `RT) : 0); // $0 if not writing
    ID MemRead <= MemRead;
    ID ALUOp <= ALUOp;
    ID MemWrite <= MemWrite;
    ID Bad <= Bad;
  end
end

// EX: EXecute stage
always @(posedge clk) if (running) begin
  case (ID ALUOp)
    `ALUAND: alu = ID s & ID src;
    `ALUOR:  alu = ID s | ID src;
    `ALUADD: alu = ID s + ID src;
    `ALUSUB: alu = ID s - ID src;
    `ALUSLT: alu = ID s < ID src;
    `ALUXOR: alu = ID s ^ ID src;
    default: alu = (ID src << 16);
  endcase

  EX alu <= alu;
  EX t <= ID t;
  EX rd <= ID rd;
  EX MemRead <= ID MemRead;
  EX MemWrite <= ID MemWrite;
  EX Bad <= ID Bad;
end
```

Color key: initialization
IF squashed ID blocked EX MEM WB
debugging

```
// MEM: data MEMory access stage
always @(posedge clk) if (running) begin
  if (EX MemRead) v = m[EX alu >> 2]; else v = EX alu;
  if (EX MemWrite) m[EX alu >> 2] <= EX t;

  MEM v <= v;
  MEM rd <= EX rd;
  MEM Bad <= EX Bad;
end

// WB: register Write Back stage
always @(posedge clk) if (running) begin
  if (MEM rd) r[MEM rd] <= MEM v;
  if (MEM Bad) halt <= 1;
end

// Reset
always @(posedge clk) if (reset) begin
  halt<=0;
  r[0]<=0;
  squash <= 0; target <= 0;
  IF ir<=`NOP; IF pc<=0;
  ID s<=0; ID t<=0; ID src<=0; ID rd<=0; ID MemRead<=0; ID ALUOp<=0; ID MemWrite<=0; ID Bad<=0;
  EX alu<=0; EX t<=0; EX rd<=0; EX MemRead<=0; EX MemWrite<=0; EX Bad<=0;
  MEM v<=0; MEM rd<=0; MEM Bad<=0;
end

`ifdef TRACE
// State-by-state trace
always @(posedge clk) if (running) begin
`define Fs      (canfwds ? (EX deps ? "EX " : (MEM deps ? "MEM " : "WB ")) : "")
`define Ft      (canfwdt ? (EX dept ? "EX " : (MEM dept ? "MEM " : "WB ")) : "")
  $display("\n    running=%1d blocked=%1d s=%s$%1d t=%s$%1d squash=%1d", running, blocked, `Fs, IF ir `RS, `Ft, IF ir `RT, squash);
  $display("IF  ir=%x pc=%1d", IF ir, IF pc);
  case (IF ir `OP)
    `RTYPE: begin
      case (IF ir `FUNCT)
        `ADDU:   $display("IF  addu $%1d,$%1d,$%1d", IF ir `RD, IF ir `RS, IF ir `RT);
        `SUBU:   $display("IF  subu $%1d,$%1d,$%1d", IF ir `RD, IF ir `RS, IF ir `RT);
        `AND:    $display("IF  and $%1d,$%1d,$%1d", IF ir `RD, IF ir `RS, IF ir `RT);
        `OR:     $display("IF  or $%1d,$%1d,$%1d", IF ir `RD, IF ir `RS, IF ir `RT);
        `XOR:    $display("IF  xor $%1d,$%1d,$%1d", IF ir `RD, IF ir `RS, IF ir `RT);
        `SLTU:   $display("IF  sltu $%1d,$%1d,$%1d", IF ir `RD, IF ir `RS, IF ir `RT);
        default: $display("IF  OP=%x RS=%1d RT=%1d RD=%1d SHAMT=%1d FUNCT=%x", IF ir `OP, IF ir `RS, IF ir `RT, IF ir `RD, IF ir `SHAMT, IF ir `FUNCT);
      endcase
    end
    `BEQ:    $display("IF  beq $%1d,$%1d,offset=$%1d", IF ir `RT, IF ir `RS, IF ir `IMM);
    `ADDIU:  $display("IF  addiu $%1d,$%1d,$%1d", IF ir `RT, IF ir `RS, IF ir `IMM);
    `SLTIU:  $display("IF  sltiu $%1d,$%1d,$%1d", IF ir `RT, IF ir `RS, IF ir `IMM);
    `ANDI:   $display("IF  andi $%1d,$%1d,$%1d", IF ir `RT, IF ir `RS, IF ir `IMM);
    `ORI:    $display("IF  ori $%1d,$%1d,$%1d", IF ir `RT, IF ir `RS, IF ir `IMM);
    `XORI:   $display("IF  xori $%1d,$%1d,$%1d", IF ir `RT, IF ir `RS, IF ir `IMM);
    `LUI:    $display("IF  lui $%1d,$%1d", IF ir `RS, IF ir `IMM);
    `LW:     $display("IF  lw $%1d,$%1d($%1d)", IF ir `RT, IF ir `IMM, IF ir `RS);
    `SW:     $display("IF  sw $%1d,$%1d($%1d)", IF ir `RT, IF ir `IMM, IF ir `RS);
    default: $display("IF  OP=%x RS=%1d RT=%1d IMM=%x", IF ir `OP, IF ir `RS, IF ir `RT, IF ir `IMM);
  endcase
  if (ID Bad) $display("ID  illegal instruction");
    else $display("ID  s=%1d t=%1d src=%1d rd=%1d MemRead=%b ALUOp=%b MemWrite=%b", ID s, ID t, ID src, ID rd, ID MemRead, ID ALUOp, ID MemWrite);
  if (EX Bad) $display("EX  illegal instruction");
    else $display("EX  alu=%1d t=%1d rd=%1d MemRead=%b MemWrite=%b", EX alu, EX t, EX rd, EX MemRead, EX MemWrite);
  if (MEM Bad) $display("MEM illegal instruction");
    else begin
      if (EX MemWrite) $display("MEM m[%1d]=%1d v=%1d rd=%1d", EX alu, EX t, MEM v, MEM rd);
        else $display("MEM v=%1d rd=%1d", MEM v, MEM rd);
    end
  if (MEM rd) $display("WB  r[%1d]=%1d", MEM rd, MEM v);
end
`endif
endmodule

// Testbench
module bench;
wire halt;
reg reset = 1;
reg clk = 0;

processor PE(halt, reset, clk);

initial begin
  #`CLKDEL clk = 1;
  #`CLKDEL clk = 0;
  reset = 0;
  while (($time < `RUNTIME) && !halt) begin
    #`CLKDEL clk = 1;
    #`CLKDEL clk = 0;
  end
end
endmodule
```

IF: Instruction Fetch stage

- Really simple…
  - Memory is **`WORD**, not byte, so address>>**2**
  - The only thing setting **IF_ir** and **IF_pc**

```
// IF: Instruction Fetch stage
always @(posedge clk) if (running && !blocked) begin
  IF_ir <= m[(squash ? target : IF_pc) >> 2];
  IF_pc <= (squash ? target : IF_pc) + 4;
end
```

ID: Instruction Decode

- Decodes the instruction
- Reads from register file **`r[]`**
- Computes **`beq`** comparison & **`target`**

```
// ID: Instruction Decode stage
always @(posedge clk) if (running && !ID_Bad) begin
  if (blocked) … else begin
    case (squashed `OP)
      `RTYPE: begin
        case (squashed `FUNCT)
          `ADDU:   begin RegDst=1; Branch=0; MemRead=0;
                   ALUOp=`ALUADD; MemWrite=0; ALUSrc=0;
                   RegWrite=1; Bad=0; end
    … endcase … endcase
    … ID_s <= s; ID_t <= t; … ID_ALUOp <= ALUOp; …
end end
```

EX: Execute stage

- Contains the ALU for integers & addresses

```
// EX: EXecute stage
always @(posedge clk) if (running) begin
  case (ID_ALUOp)
    `ALUAND: alu = ID_s & ID_src;
    `ALUOR:  alu = ID_s | ID_src;
    `ALUADD: alu = ID_s + ID_src;
    `ALUSUB: alu = ID_s - ID_src;
    `ALUSLT: alu = ID_s < ID_src;
    `ALUXOR: alu = ID_s ^ ID_src;
    default: alu = (ID_src << 16);
  endcase
  EX_alu <= alu;
  EX_t <= ID_t;
  EX_rd <= ID_rd;
  EX_MemRead <= ID_MemRead;
  EX_MemWrite <= ID_MemWrite;
  EX_Bad <= ID_Bad;
end
```

MEM: MEMory access

- Does a memory read or write
- Uses **`EX_MemRead`** for both read and mux **`v`**

```
// MEM: data MEMory access stage
always @(posedge clk) if (running) begin
  if (EX_MemRead) v = m[EX_alu >> 2]; else v = EX_alu;
  if (EX_MemWrite) m[EX_alu >> 2] <= EX_t;

  MEM_v <= v;
  MEM_rd <= EX_rd;
  MEM_Bad <= EX_Bad;
end
```

WB: Write Back

- Writes result into register…
  - Register $0 is read only
  - Not writing? Say we write register 0…
- An instruction isn't done until it's here, so this is where halting really happens

```
// WB: register Write Back stage
always @(posedge clk) if (running) begin
  if (MEM_rd) r[MEM_rd] <= MEM_v;
  if (MEM_Bad) halt <= 1;
end
```

Running?

- Enables normal operation of stages
- Not running normally if:
  - Halted
  - There’s a reset in progress; reset writes into stuff it doesn’t own, so we need owners disabled

```
// Running state?
wire running;
assign running = ((!halt) && (!reset));
```

Squashed?

- For **`beq`**, predict not taken, so **`IF_pc+4`**
- If we were right, no bubble
- If wrong, squash fetched instructions
  - No side-effects to undo yet
  - Convert into **`` `NOP ``** to prevent future s-e

```
`define NOP `OR   // Null OPeration is or $0,$0,$0

// Squash instruction fetched on a mispredicted branch
wire `INST squashed;
assign squashed = (squash ? `NOP : IF_ir);
```

Data Dep. / Forwarding

- Need `$rs`, `$rt` unless either is `$0`
- Desired value could be in EX, MEM, or WB
- Forward priority to EX over MEM over WB
- Can’t forward `lw` value from EX stage!

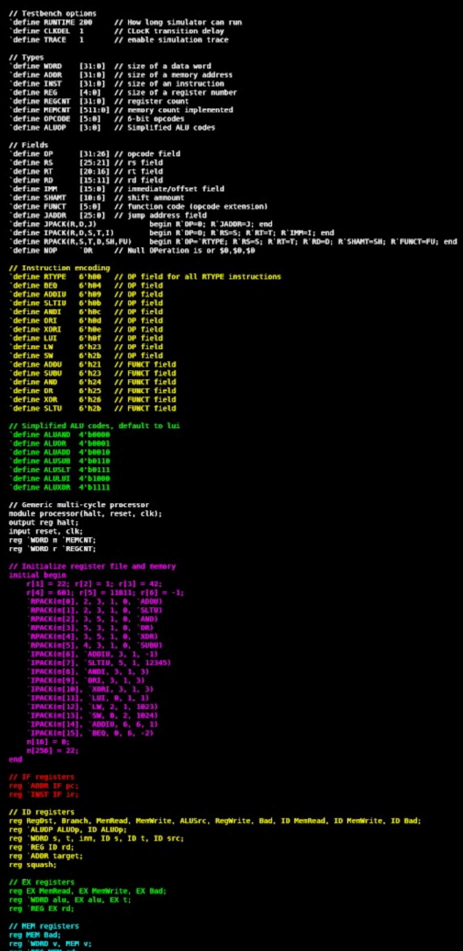
```
// Are we blocked by a dependence on rs or rt?
// Also handles all forwarding
…
assign canfwds = ID_needs && ((EX_deps && !MemRead) ||
                 ((!EX_deps) && (MEM_deps || WB_deps)));
assign canfwdt = ID_needt && ((EX_dept && !MemRead) ||
                 ((!EX_dept) && (MEM_dept || WB_dept)));
assign fwds = (EX_deps ? alu : (MEM_deps ? v : MEM_v));
assign fwdt = (EX_dept ? alu : (MEM_dept ? v : MEM_v));
assign dep = ((ID_needs && (EX_deps||MEM_deps||WB_deps)) ||
          (ID_needt && (EX_dept || MEM_dept || WB_dept)));
assign blocked = (dep && ((ID_needs && !canfwds) ||
                          (ID_needt && !canfwdt)));
```

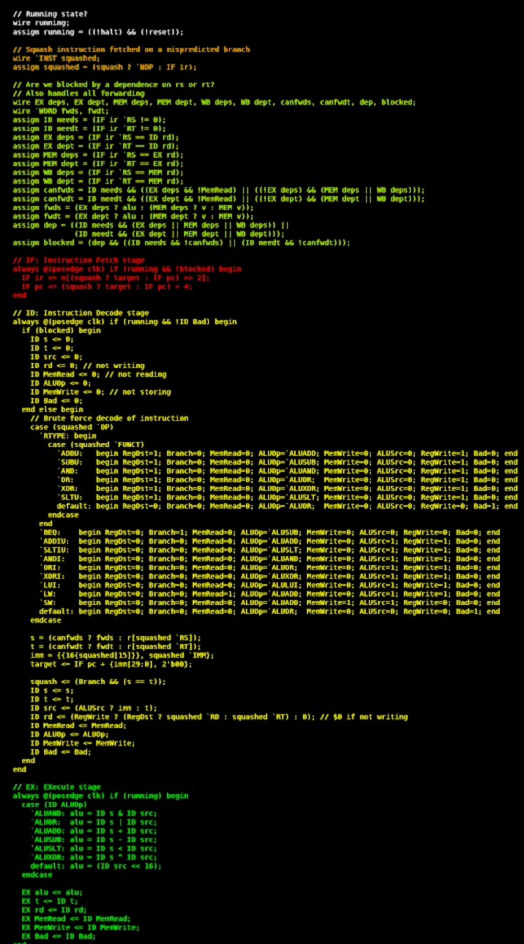
Simulation Tracing

- More (complex!) parallel-executing stuff:

```
// State-by-state trace
`ifdef   TRACE
always @(posedge clk) if (running) begin
  …
  $display("IF ir=%x pc=%1d", IF_ir, IF_pc);
  case (IF_ir `OP)
    `RTYPE: begin
      case (IF_ir `FUNCT)
        `ADDU:  $display("IF  addu $%1d,$%1d,$%1d",
                 IF_ir `RD, IF_ir `RS, IF_ir `RT); …
  endcase endcase
  if (ID_Bad) $display("ID  illegal instruction");
    else $display("ID  s=%1d t=%1d src=%1d rd=%1d
      MemRead=%b ALUOp=%b MemWrite=%b", ID_s, ID_t,
      ID_src, ID_rd, ID_MemRead, ID_ALUOp, ID_MemWrite);
   …
end
`endif
endmodule
```

Pipelined Verilog Version





Color key: initialization
IF squashed ID blocked EX MEM WB
debugging

Three Verilog Implementations

- **Multi-cycle** MIPS, multiple CPI:

  http://aggregate.org/EE380/multiv.html

- **Single-cycle**, 1 CPI, but slow clock:

  http://aggregate.org/EE380/onebeq.html

- **Pipelined**, fast clock, ~1 CPI throughput:

  http://aggregate.org/EE380/pipe.html