

Single-Cycle Design

CPE380, Spring 2026

Hank Dietz

<http://aggregate.org/hankd/>

A Single-Cycle MIPS Design

- Process one instruction at a time...
- The **multi-cycle** MIPS we built before:

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

- Each instruction takes **multiple clock cycles**
- **Minimal hardware** & 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**

A Single-Cycle MIPS Design

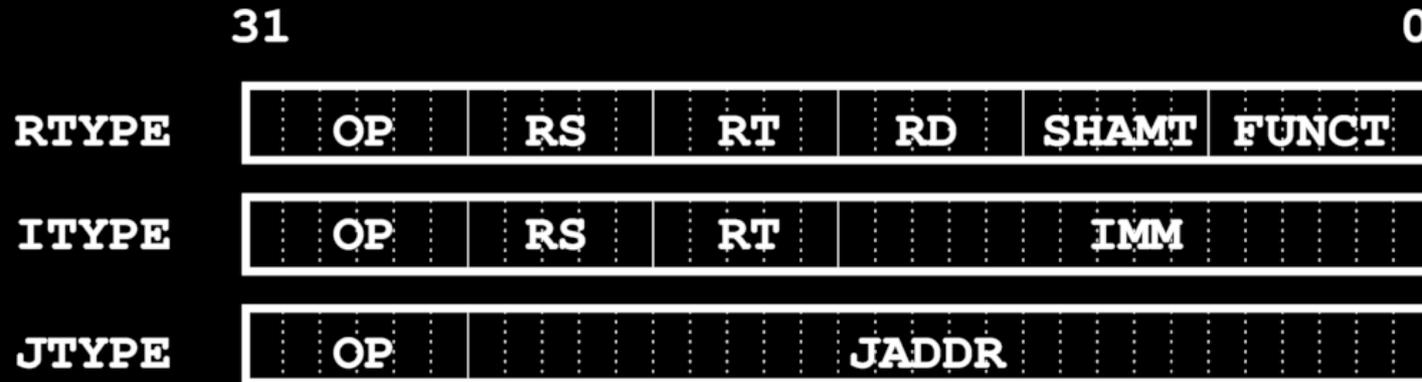
- This is **not** a design you'd really build...
it's a step toward building a **pipelined design**
- We're not going to design it all at once
 - Can **incrementally design & test**
 - Start by implementing one instruction
 - Handle an instruction sequence
 - Keep adding instruction implementations...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

Types Of Things In MIPS

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

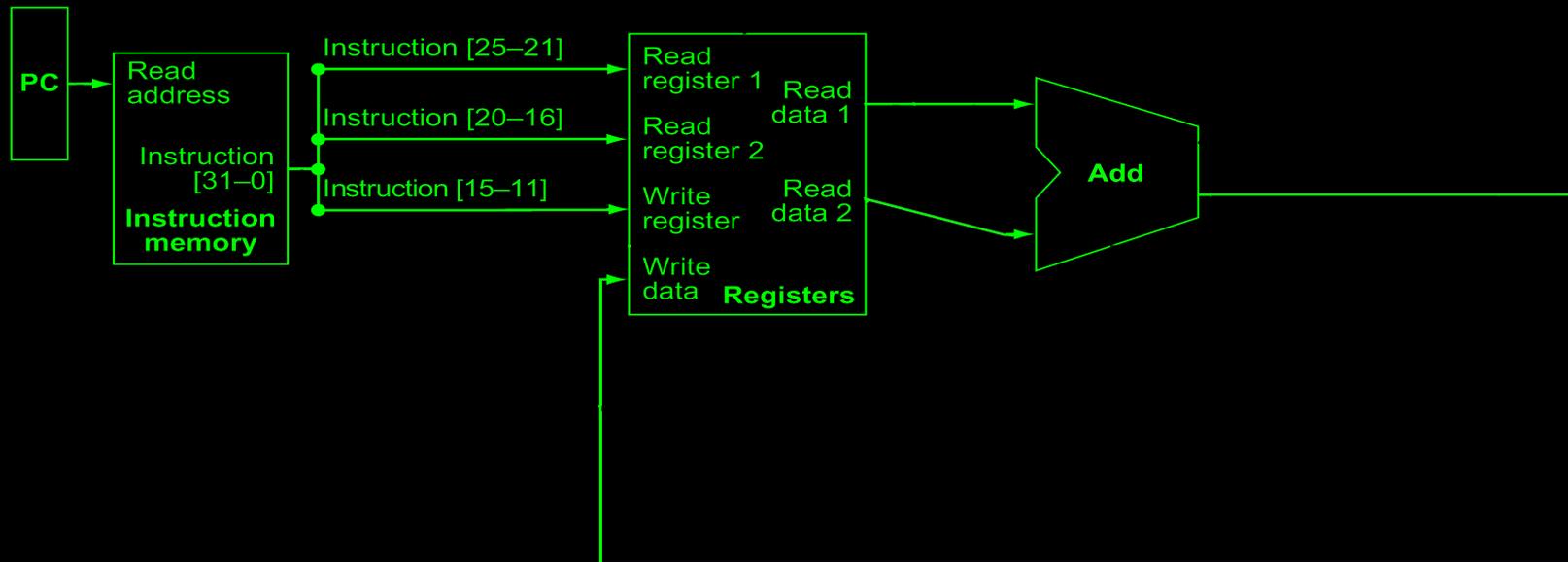
```
// Types
`define WORD      [31:0] // size of a data word
`define ADDR     [31:0] // size of a memory address
`define INST     [31:0] // size of an instruction
`define REG      [4:0]  // size of a register number
`define REGCNT   [31:0] // register count
`define MEMCNT   [511:0] // memory count implemented
`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,0,J) begin R`OP=0; R`JADDR=J; end
`define IPACK(R,0,S,T,I) begin R`OP=0; 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
```

The Complete MIPS Processor*

**that only can execute one addu*

- Can run it here:

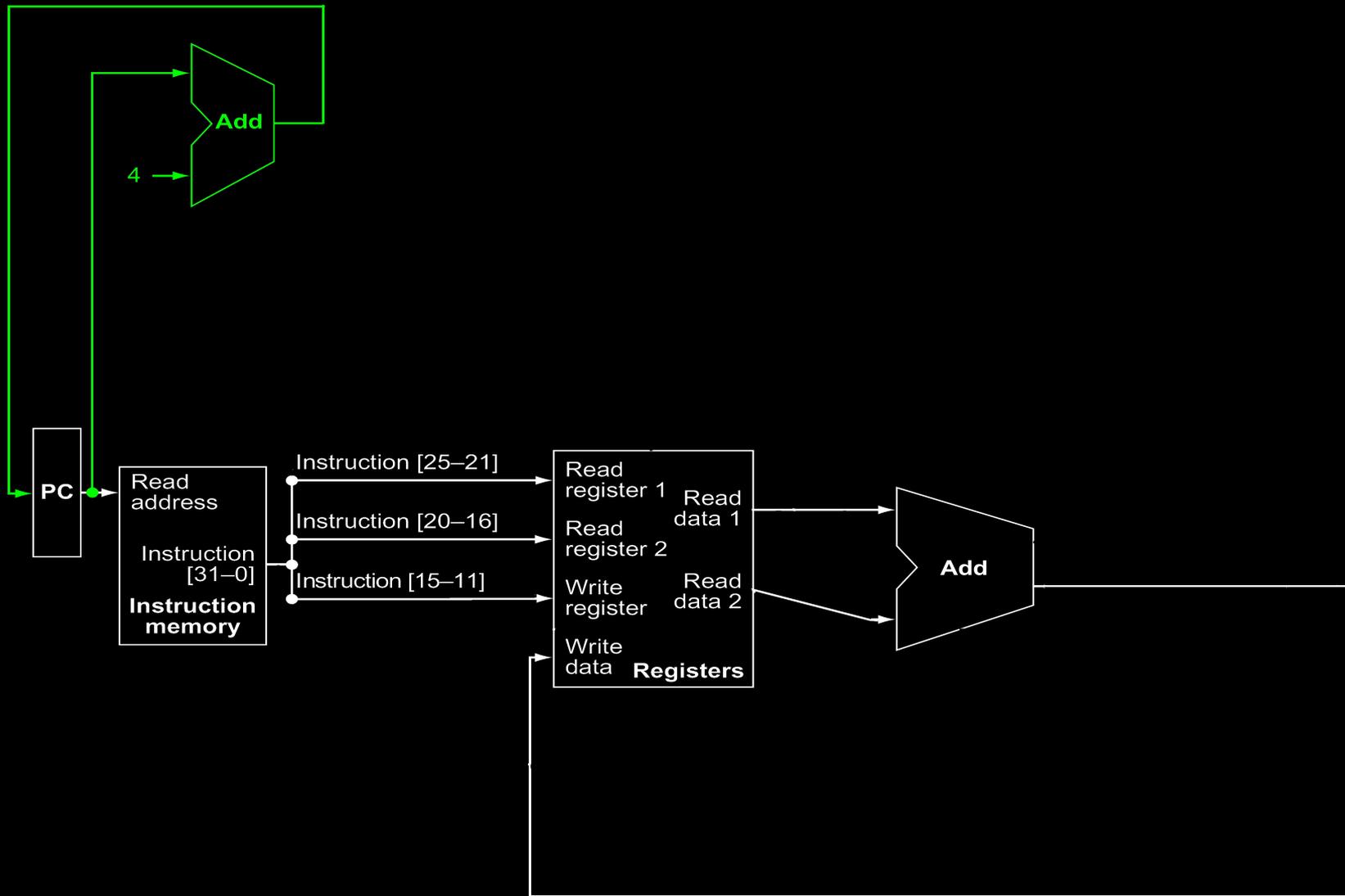
<http://aggregate.org/EE380/oneaddu.html>

- Impressed?

The 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 an entire sequence of addu?



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:

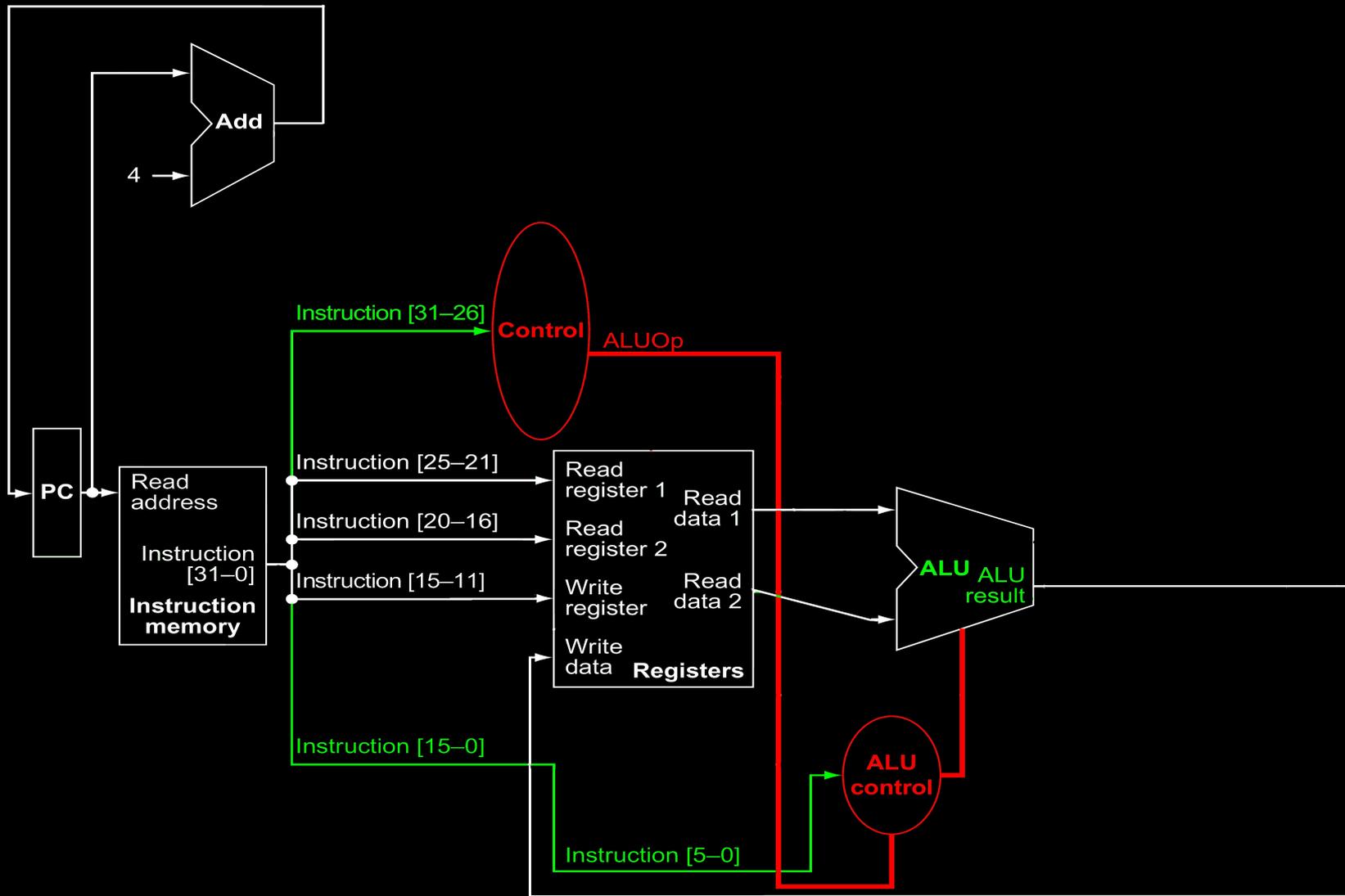
<http://aggregate.org/EE380/oneaddus.html>

Other RTYPE Instructions?

- There are lots of instructions like `addu`:

<code>addu</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs + \$rt$
<code>sltu</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs < \$rt$
<code>and</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs \& \$rt$
<code>or</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs \$rt$
<code>xor</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs \wedge \$rt$
<code>subu</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs - \$rt$

- Handle them all the same, except in ALU



addu \$rd,\$rs,\$rt
and \$rd,\$rs,\$rt

subu \$rd,\$rs,\$rt
or \$rd,\$rs,\$rt

slt \$rd,\$rs,\$rt
xor \$rd,\$rs,\$rt

Decoding RTYPE Instructions

```
// 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 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)
        `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 Instructions

- 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:

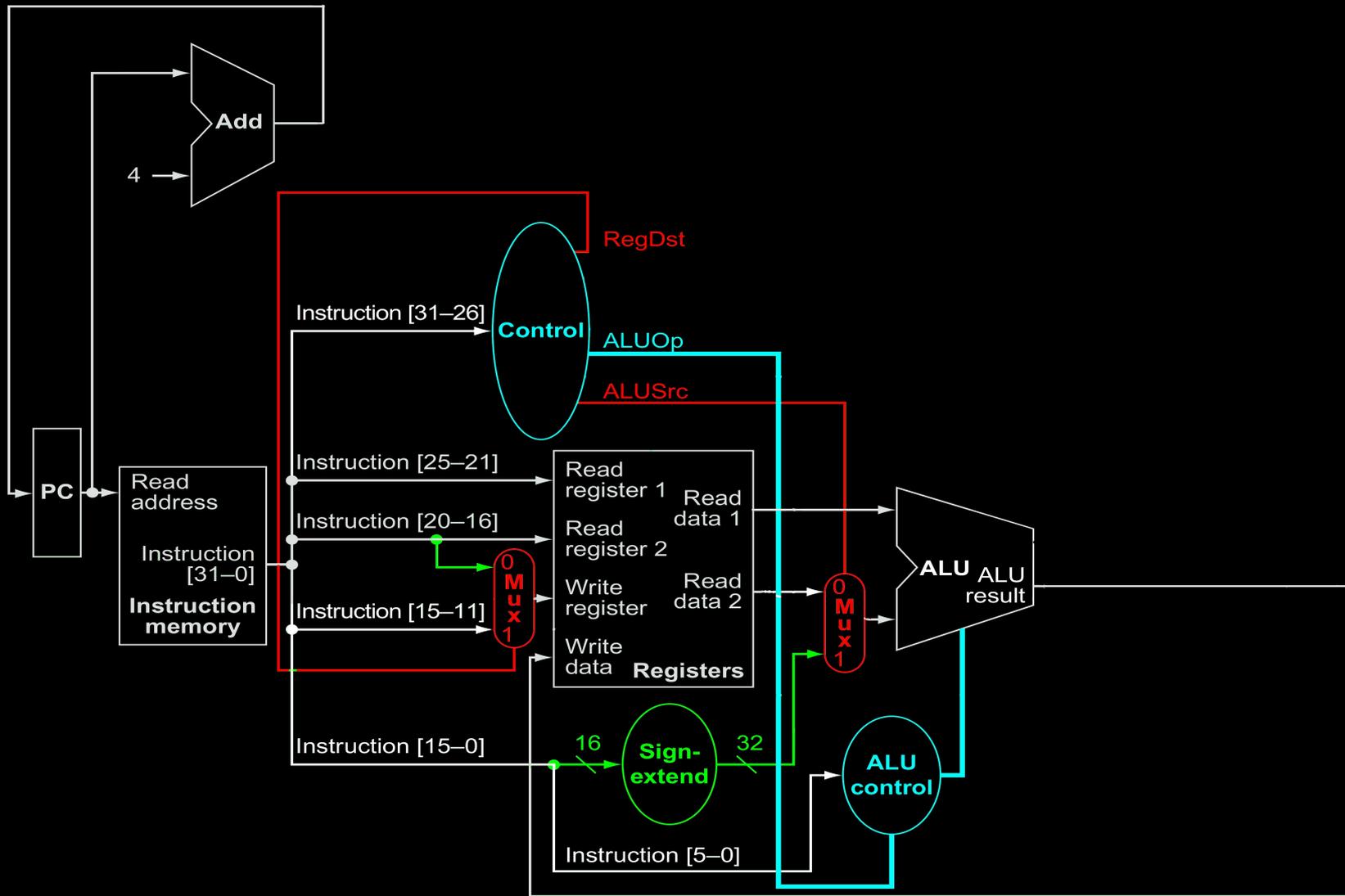
<http://aggregate.org/EE380/onertypes.html>

How About Immediates?

- Immediates use the ALU similarly...

<code>addiu</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs + imm$
<code>sltiu</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs < imm$
<code>andi</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs \& imm$
<code>ori</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs imm$
<code>xori</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs \wedge imm$
<code>lui</code>	<code>\$rt, imm</code>	$\$rt = imm \ll 16$

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



addiu \$rt,\$rs,imm
andi \$rt,\$rs,imm

sltiu \$rt,\$rs,imm
ori \$rt,\$rs,imm

lui \$rt,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 Instructions

```
// 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
```

For Both 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 also 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>

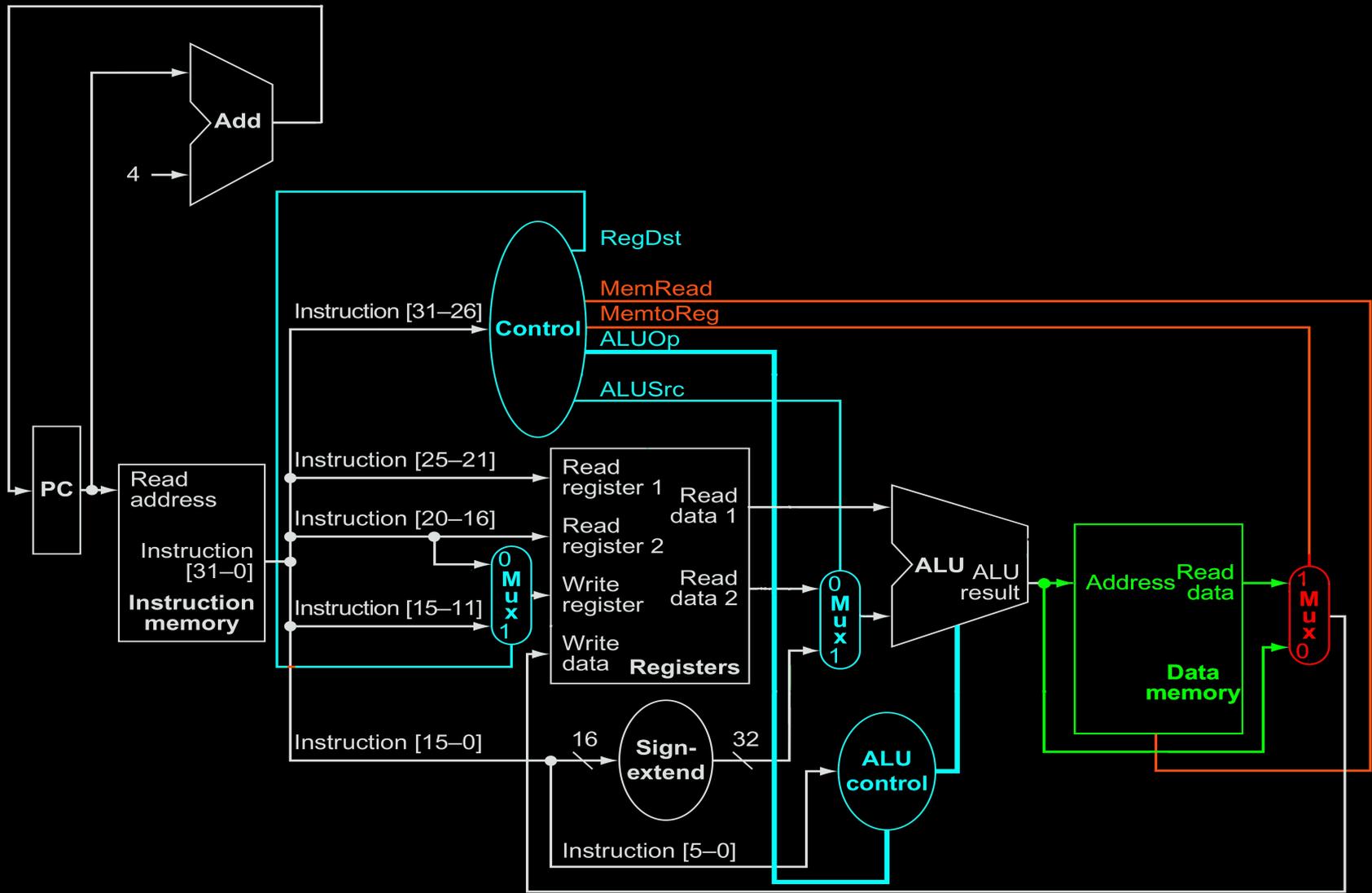
Load From Memory

- There's only one load word instruction:

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

```
$rt = memory[imm + $rs]
```

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



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/one1w.html>

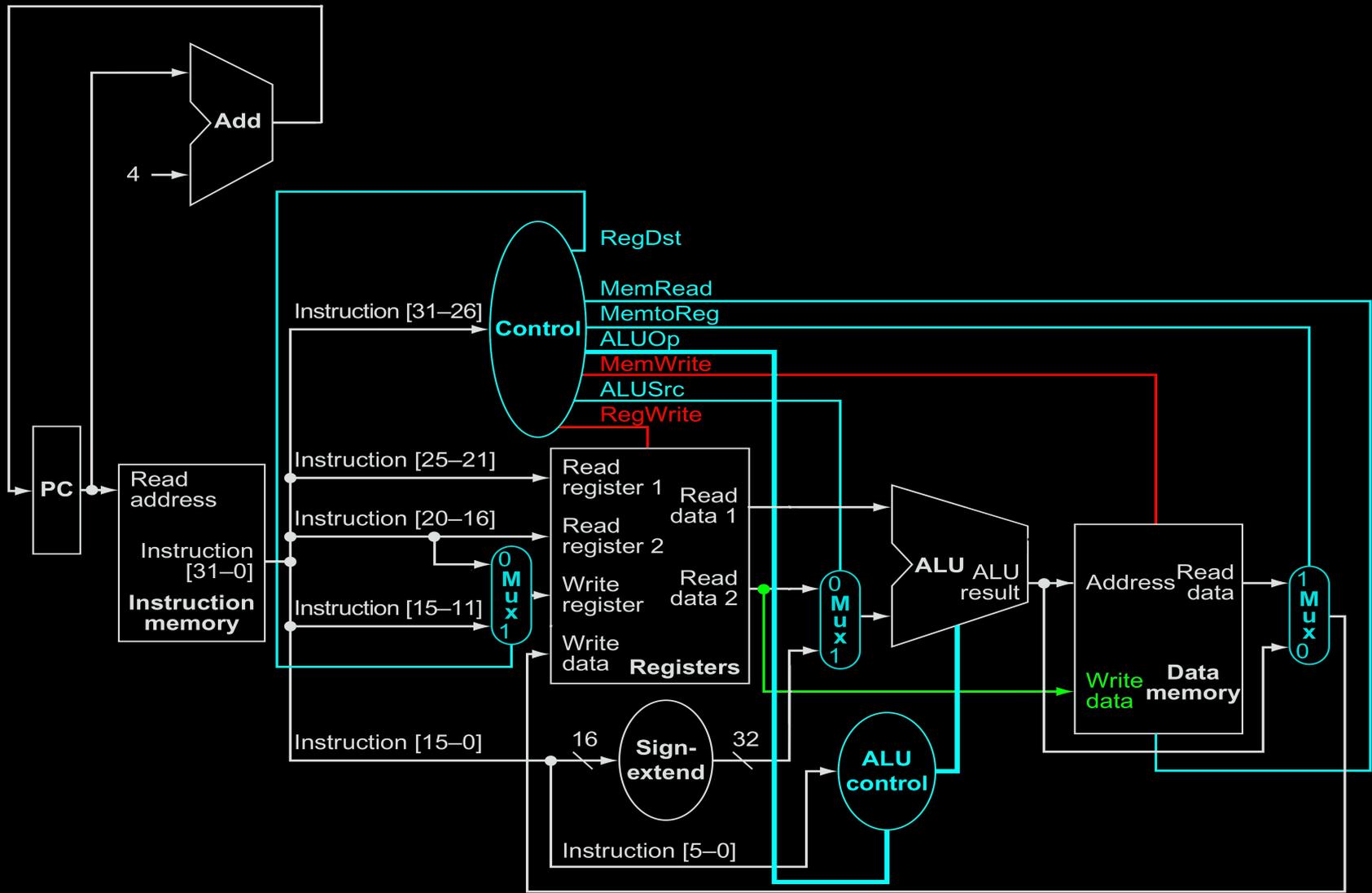
Store To Memory

- There's only one store word instruction:

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

```
memory[imm + $rs] = $rt
```

- It's sort-of an `lw...`



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?

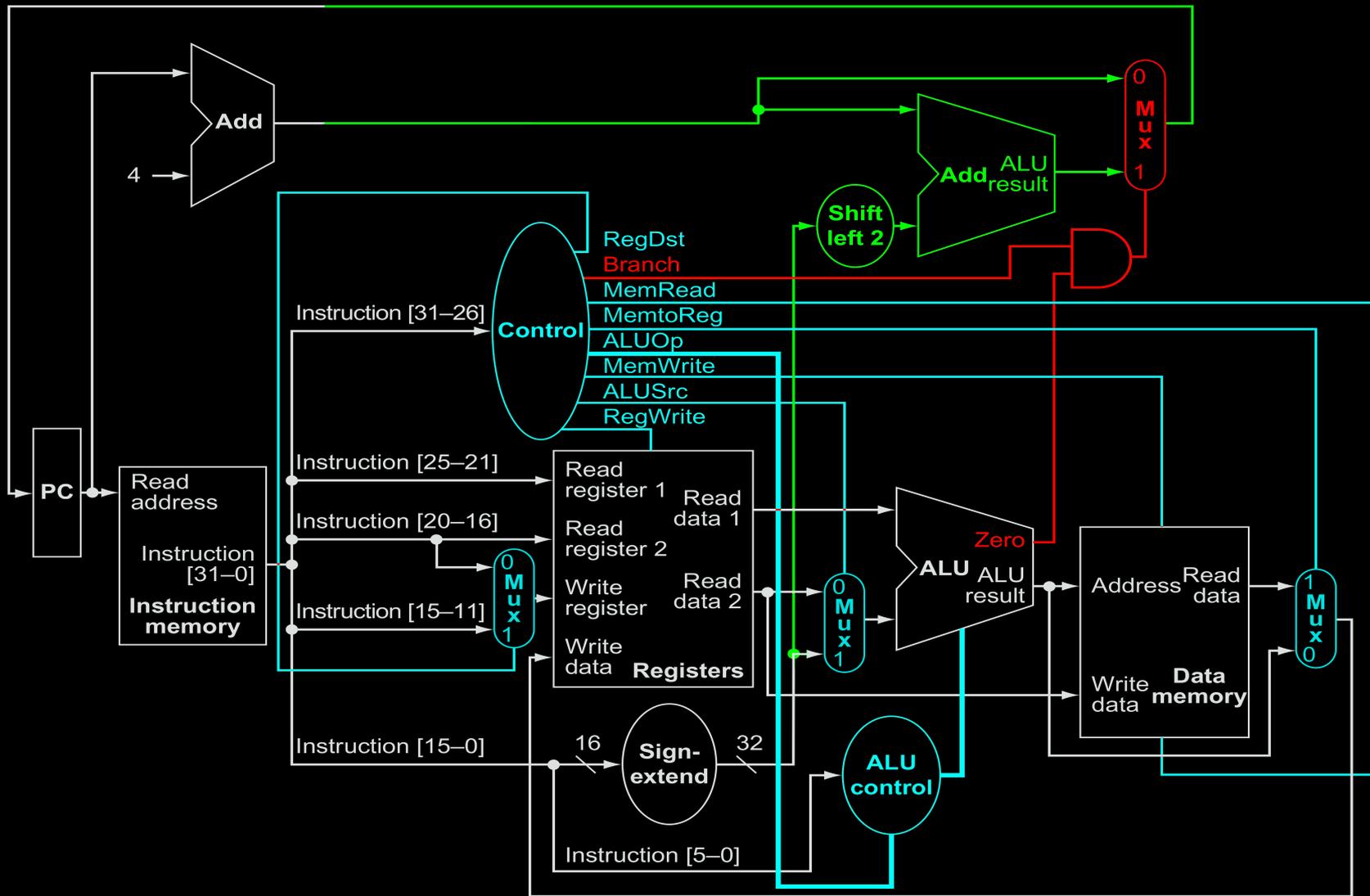
- How about a branch equals?

`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

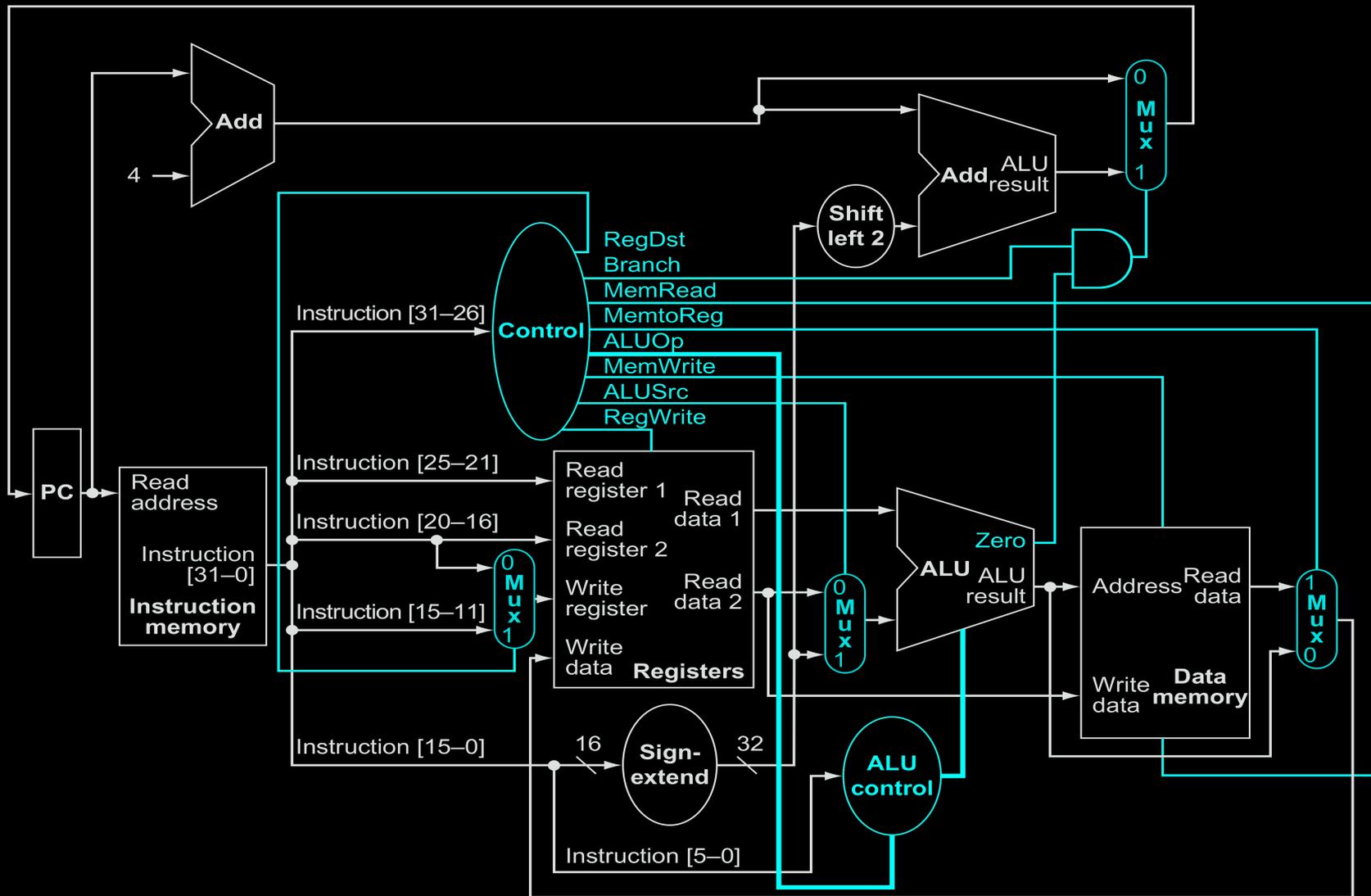
```
assign Shiftleft2 = (Signextend << 2);  
assign BranchAdd = (PCAdd + Shiftleft2);
```

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

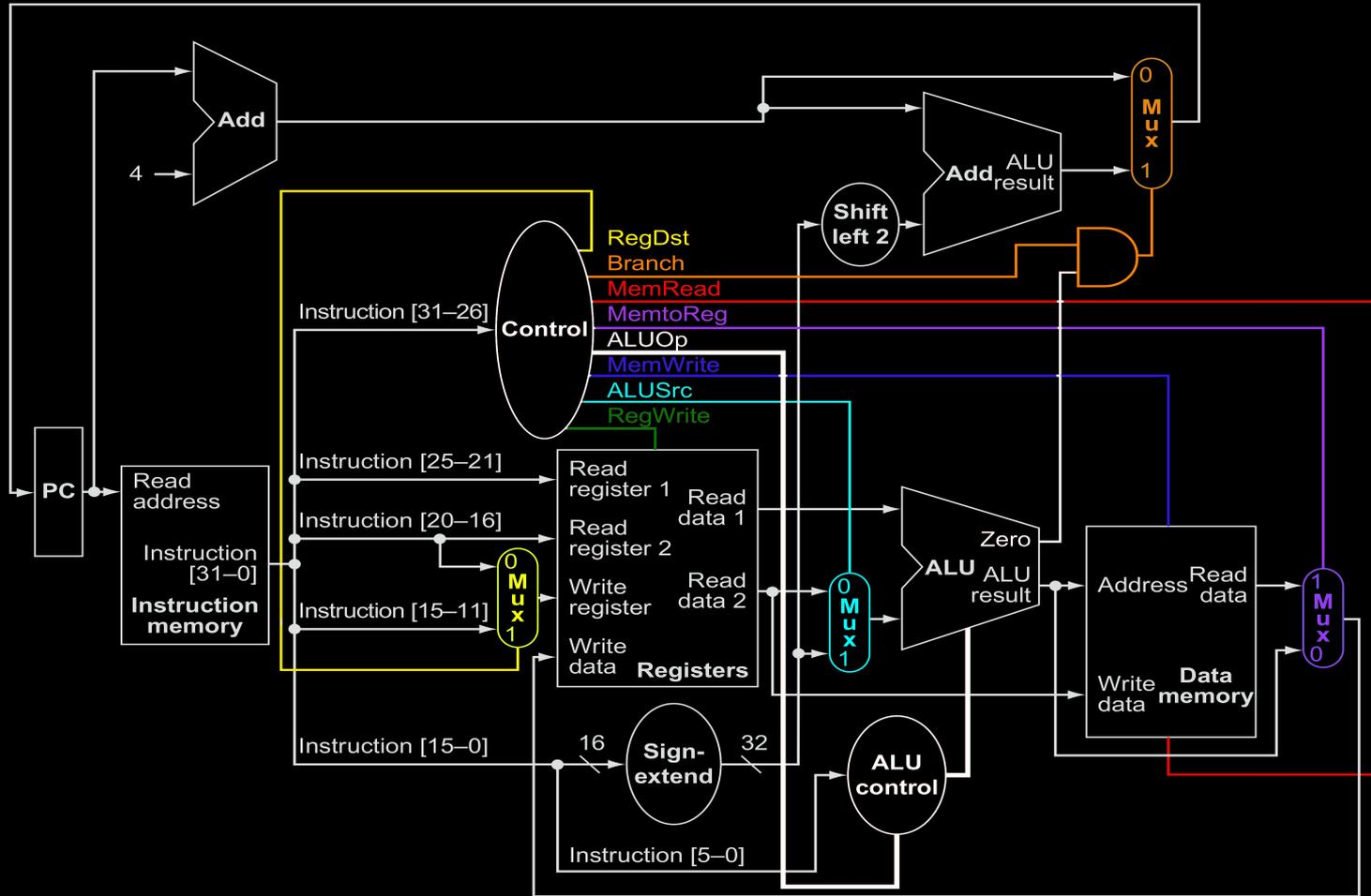
```
assign ALUSrc = ((ir `OP != `RTYPE) && (ir `OP != `BEQ));  
assign zero = (res == 0);  
assign Branch = (ir `OP == `BEQ);  
assign BranchZeroMux = ((Branch & Zero) ? BranchAdd:PCAdd);
```

- Can run it here:

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



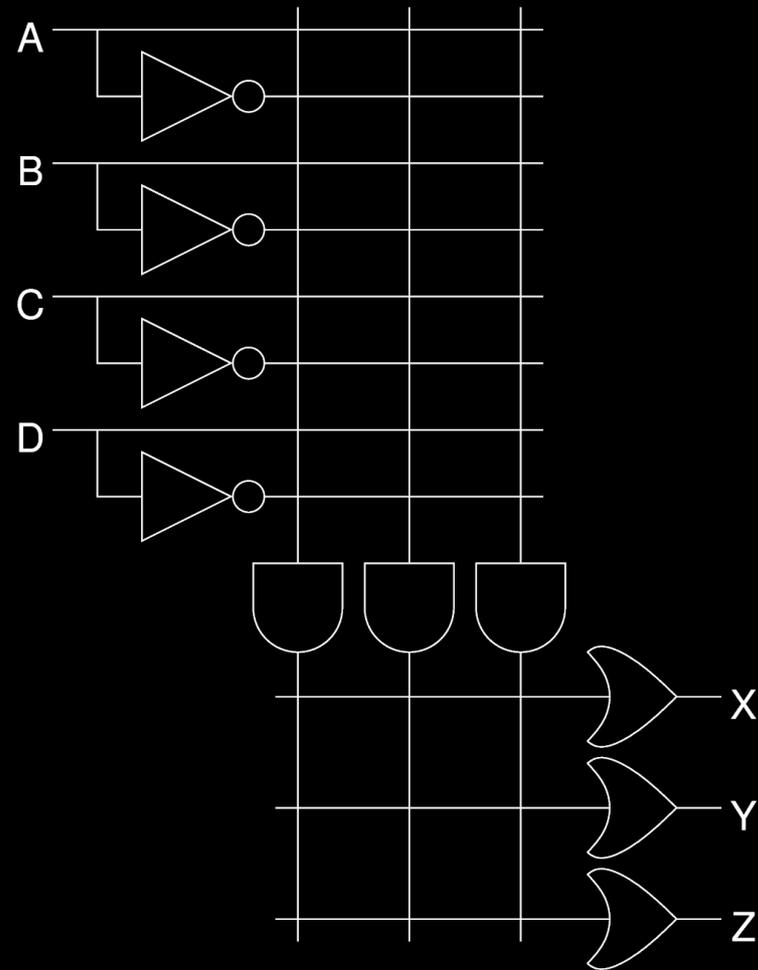
can incrementally add paths for more instructions
 # always try to reuse as much hardware as possible



	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

Logic Implementation Choices

- **Random logic:**
fast, small
(used here)
- **ROM:**
easy to change
- **Programmable Logic Array (PLA):**
balanced...



Summary

Nobody builds a machine like this!

This is a step in **designing the pipelined version.**