Processor Design in Three Acts

Act I: A single-cycle CPU

Foreshadowing

- Act I: A Single-cycle Processor
  - Simplest design – Not how many real machines work (maybe some deeply embedded processors)
  - Figure out the basic parts; what it takes to execute instructions
- Act II: A Multi-Cycle Processor
- Act III: Pipeline Processor
  - This is how many real machines work
  - Exploit parallelism by executing multiple instructions at once.

Target ISA

- We will focus on part of MIPS
  - Enough to run into the interesting issues
  - Memory operations
  - A few arithmetic/Logical operations (Generalizing is straightforward)
  - BEQ and J
- You should be able to extend it to handle other instructions
  - You will do this in 141L.

Basic Steps

- Fetch an instruction from the instruction store
- Decode it
  - What does this instruction do?
- Gather inputs
  - From the register file
  - From memory
- Perform the operation
- Write back the outputs
  - To register file or memory
- Determine the next instruction to execute

The MIPS core subset

- Arithmetic ops
  - `add rd, rs, rt`
  - `sub, and, or, slt`
  - "R-Type"
- RTL
  - PC = PC + 4
  - REG[rd] = REG[rs] op REG[rt]
- Format

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6

The MIPS core subset

- Immediate Arithmetic ops
  - `add rd, rs, imm`
  - `sub, subu, addu, and, or, slt`
  - "I-Type"
- RTL -- signed
  - PC = PC + 4
  - REG[rd] = REG[rs] op SignExtImm
- RTL -- unsigned
  - PC = PC + 4
  - REG[rd] = REG[rs] op ZeroExtImm
- Format

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	imm
# bits	6	5	5	16

The MIPS core subset

- Ld/St
  - `lw rt, (imm)rs`
  - `sw rt, (imm)rs`
- **RTL**
  - `PC = PC + 4`
  - `Load:REG[rt] = MEM[signextendImm + REG [rs]]`
  - `PC = PC + 4`
  - `Store: MEM[signextendImm + REG[rs]] = REG[rt]`

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	immediate
# bits	6	5	5	16

The MIPS core subset

- Branch
  - Beq rs, rt, imm
  - I-type
- RTL
  - PC = (REG[rs] == REG[rt]) ? PC + SignExtImmediate : PC + 4;
- Format

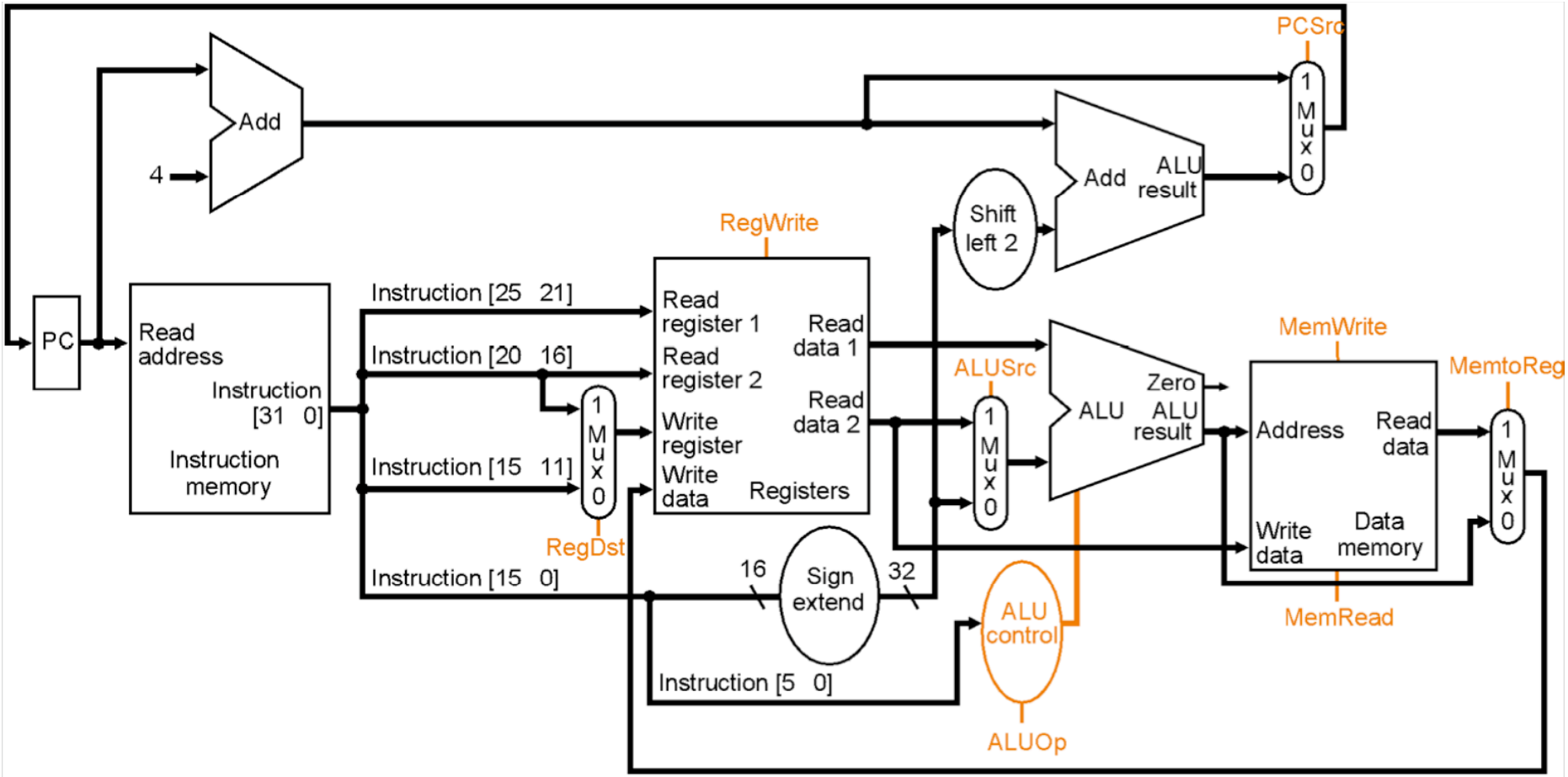
bits	31:26	25:21	20:16	15:0
name	op	rs	rt	displacement
# bits	6	5	5	16

The Processor Design Algorithm

- Once you have an ISA…
- Design/Draw the datapath
  - Identify and instantiate the hardware for your architectural state
  - Foreach instruction
    - Simulate the instruction
    - Add and connect the datapath elements it requires
    - Is it workable? If not, fix it.
- Design the control
  - Foreach instruction
    - Simulate the instruction
    - What control lines do you need?
    - How will you compute their value?
    - Modify control accordingly
    - Is it workable? If not, fix it.
- We will do this for the core subset now.
- You will do this your ISA in 141L.

The complete datapath (without jumps)

(after derivation on the blackboard)

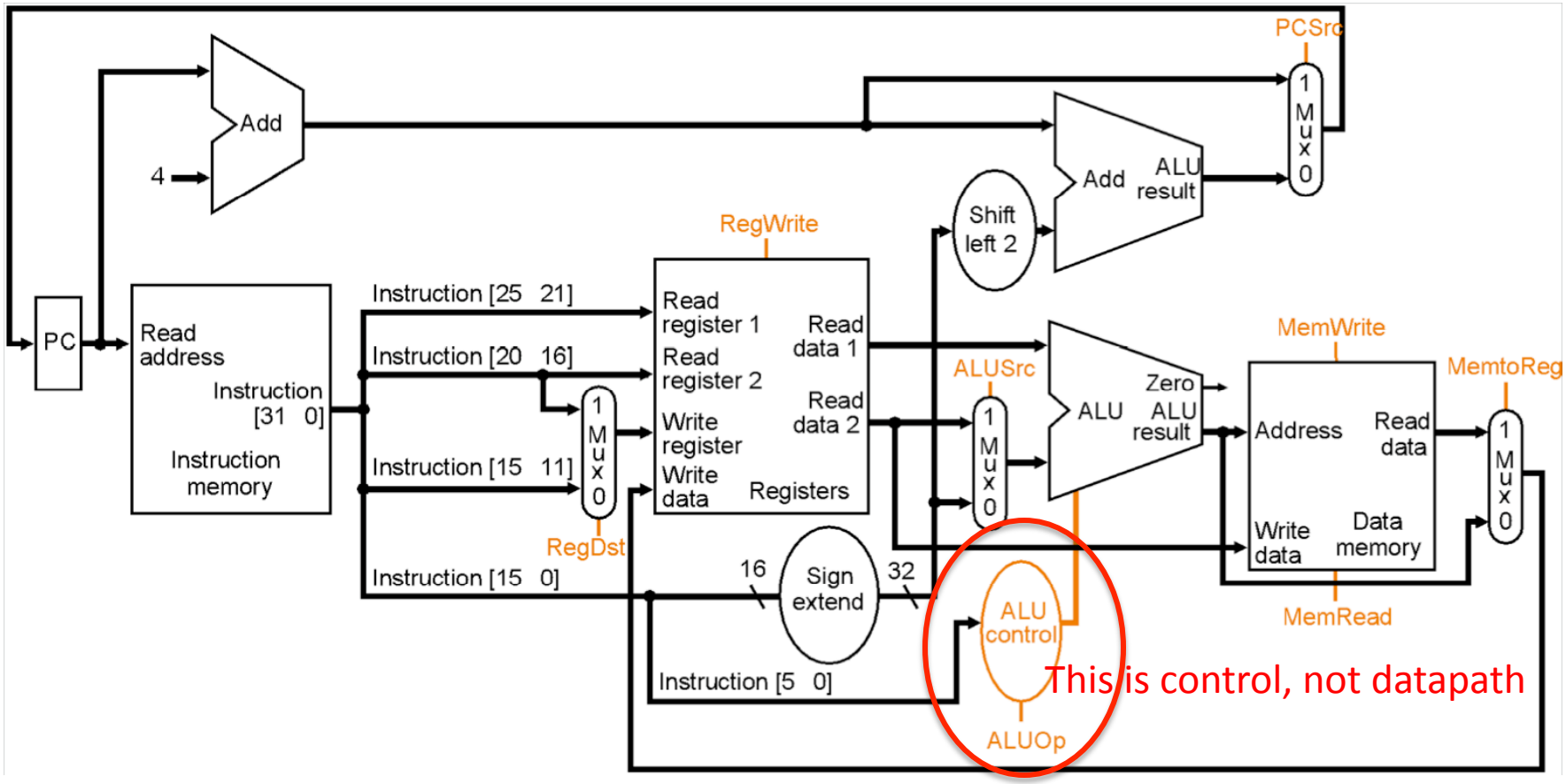


Then, code up datapath in structural verilog

```
wire [31:0] pc_plus4, branch_targ, pc_out;

wire pc_sel;


rMux2#(32) pc_mux
  ( .in0 (pc_plus4),
    .in1 (branch_targ),
    .sel (pc_sel),
    .out (pc_out));


Etc…
```

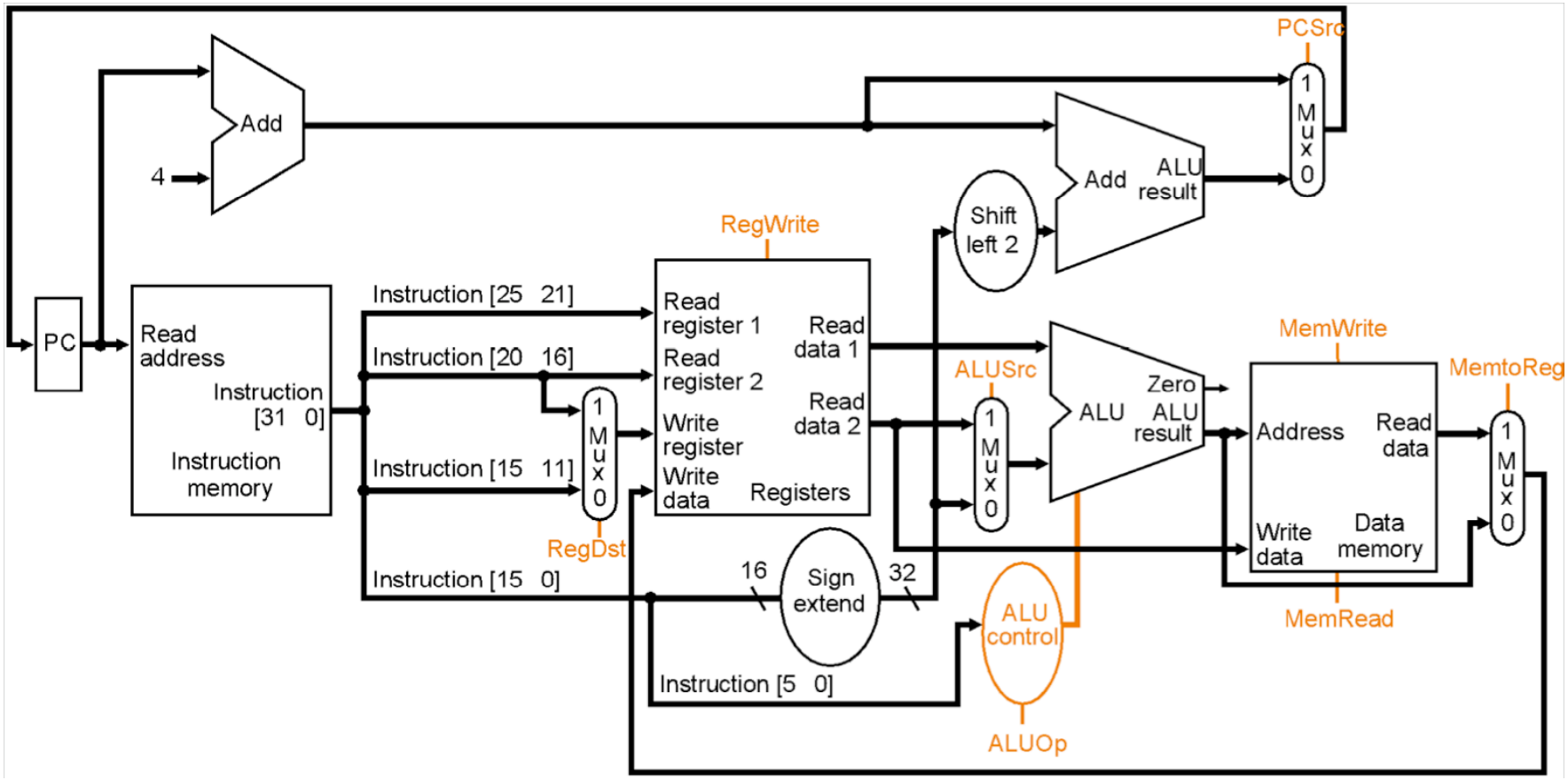
Control Signals (Control -> Datapath)

Signal	== 0	== 1
RegDst	Write to rd	Write to rt
RegWrite	Register writes suppressed	Register writes occur
ALUSrc	2nd ALU input is R[rd]	2nd ALU input is the immediate
ALUop	Multiple bits; value determines the operation the ALU will perform.	



Control Signals (Control -> Datapath)

Signal	== 0	== 1
PCSrc	PC <= PC + 4	PC <= PC + 4 + immediate
MemRead	Do not read data memory	Perform read at address
MemWrite	Do not write data memory	Perform write at address
MemtoReg	Present ALU result to register file for write	Present ALU result to register file for write.



Computing Control Signals

- hard part is to not make careless mistakes
- important to structure the code to avoid mistakes

```
`define LW    32'b100011_?????_?????_?????_?????_??????
`define SW    32'b101011_?????_?????_?????_?????_??????
`define ADDIU 32'b001001_?????_?????_?????_?????_??????
`define BNE   32'b000101_?????_?????_?????_?????_??????
```

```
reg rf_wen;  // aka RegWrite;

always @(*)
  unique casez (Instruction)
      `LW:    rf_wen = 1'b1;
      `SW:    rf_wen = 1'b0;
      `BNE:   rf_wen = 1'b0;
      `ADDIU: rf_wen = 1'b1;
      …
      default: rf_wen = 1'b0;
  endcase
```

```
reg mem_wen;  // aka MemWrite;

always @(*)
  unique casez (Instruction)
      `SW:    mem_wen = 1'b1;
      …
      default: mem_wen = 1b'0;
  endcase
```

A Single-cycle Processor

- Performance refresher

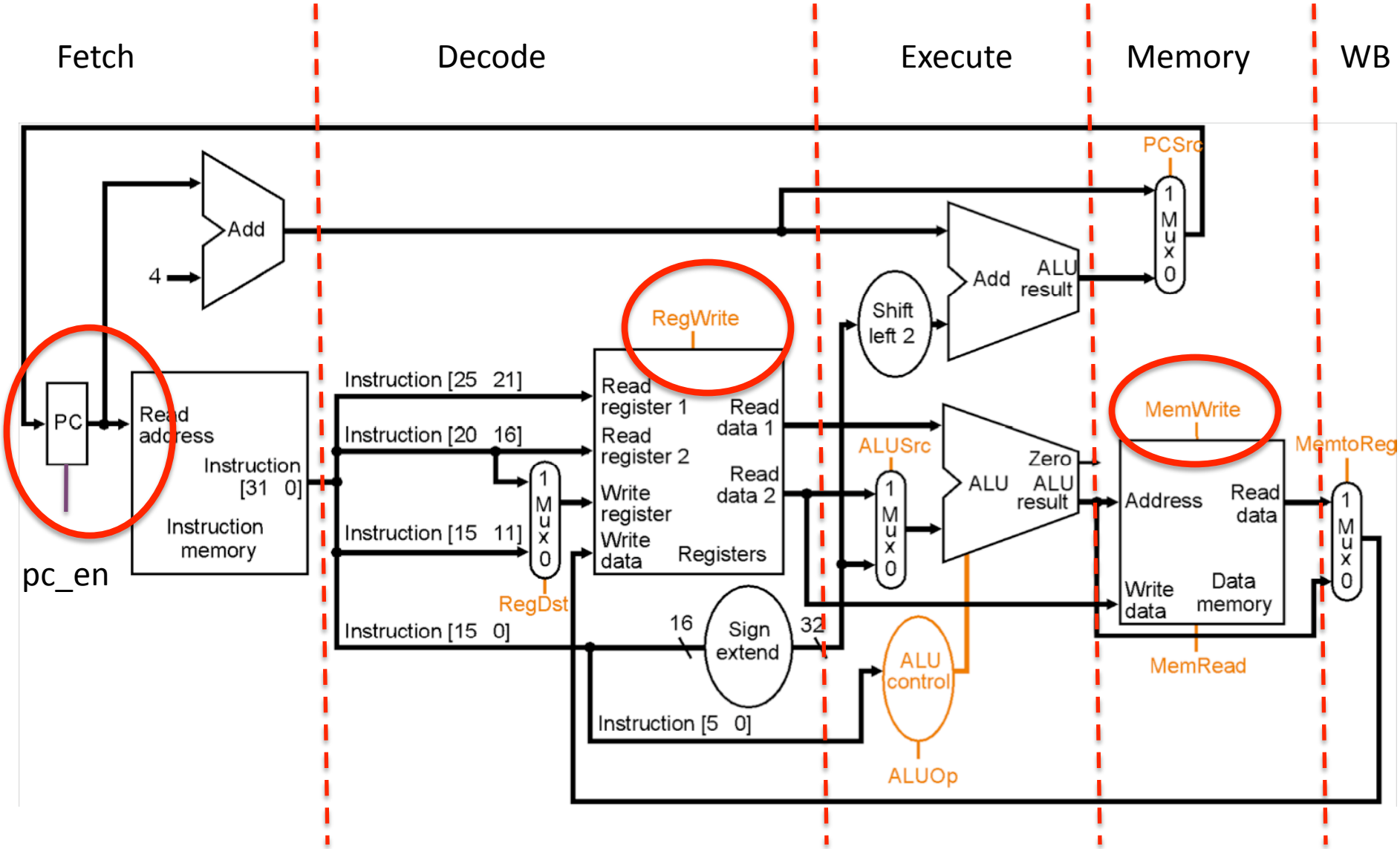
$$ET = IC * CPI * CT$$

- Single cycle ⇒ CPI == 1; That sounds great!
- But maybe we can tune CT more?

Act II: Multicycle Pipeline

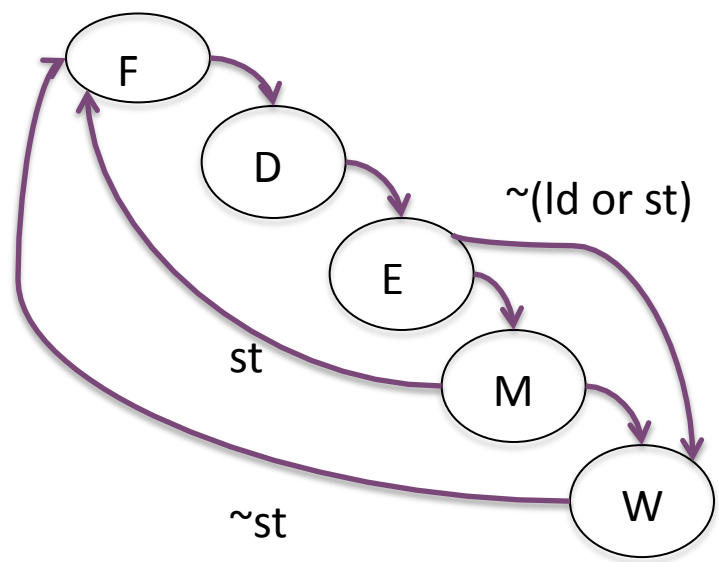
- Insight: Different instructions do not need all of the stages of execution. ALU instructions have 4 ns propagation delay and LD instructions have 5 ns. We need a way to selective spend 4 ns or 5 ns on the instruction
- Idea: We could use a clock which is GCD(4,5) = 1 ns; and then take either 4 or 5 cycles.
  - CT = CT/5.
  - 4 < IPC < 5. => 4 <= CT*IPC <= 5
  - Versus CT=5 IPC=1 => CT*IPC = 5 of single-cycle pipeline.
- Have a control state machine that varies # of cycles spent on each instruction.

Multicycle Pipeline



Multicycle Verilog

(with a little Synthesizable SystemVerilog)

```
typedef enum { sFetch, sDecode, sExecute, sMemory, sWriteback } state_s;
state_s substate_reg, substate_next;

always @(posedge clk)
   substate_reg <= reset ? sFetch : substate_next;

always @(*)
 unique case (substate_reg)
      sFetch:     substate_next = sDecode;
      sDecode: substate_next = sExecute;
      sExecute:
                    unique casez (instr_reg)
                        `SW: `LW: substate_next = sMemory;
                        default:    substate_next = sWriteback;
                    endcase
     sMemory:
                    unique casez(instruction)
                        `LW:       substate_next = sWriteback;
                         default: substate_next = sFetch;
                    endcase
     sWriteback: substate_next = sFetch;
     default: substate_next = sFetch;
 endcase
```

```
always @(posedge clk)
   substate_reg <= reset ? sFetch: substate_next;

always @(*)
 begin
   pc_en       = (substate_next == sFetch);
   mem_wen = (substate_reg == sMemory)
                    & (substate_next == sFetch);
   rf_wen      =  (substate_next = sWriteBack);
 end
```