CSCE 611:
Designing a Pipelined MIPS Processor Core

Instructor: Jason D. Bakos

UNIVERSITY OF SOUTH CAROLINA

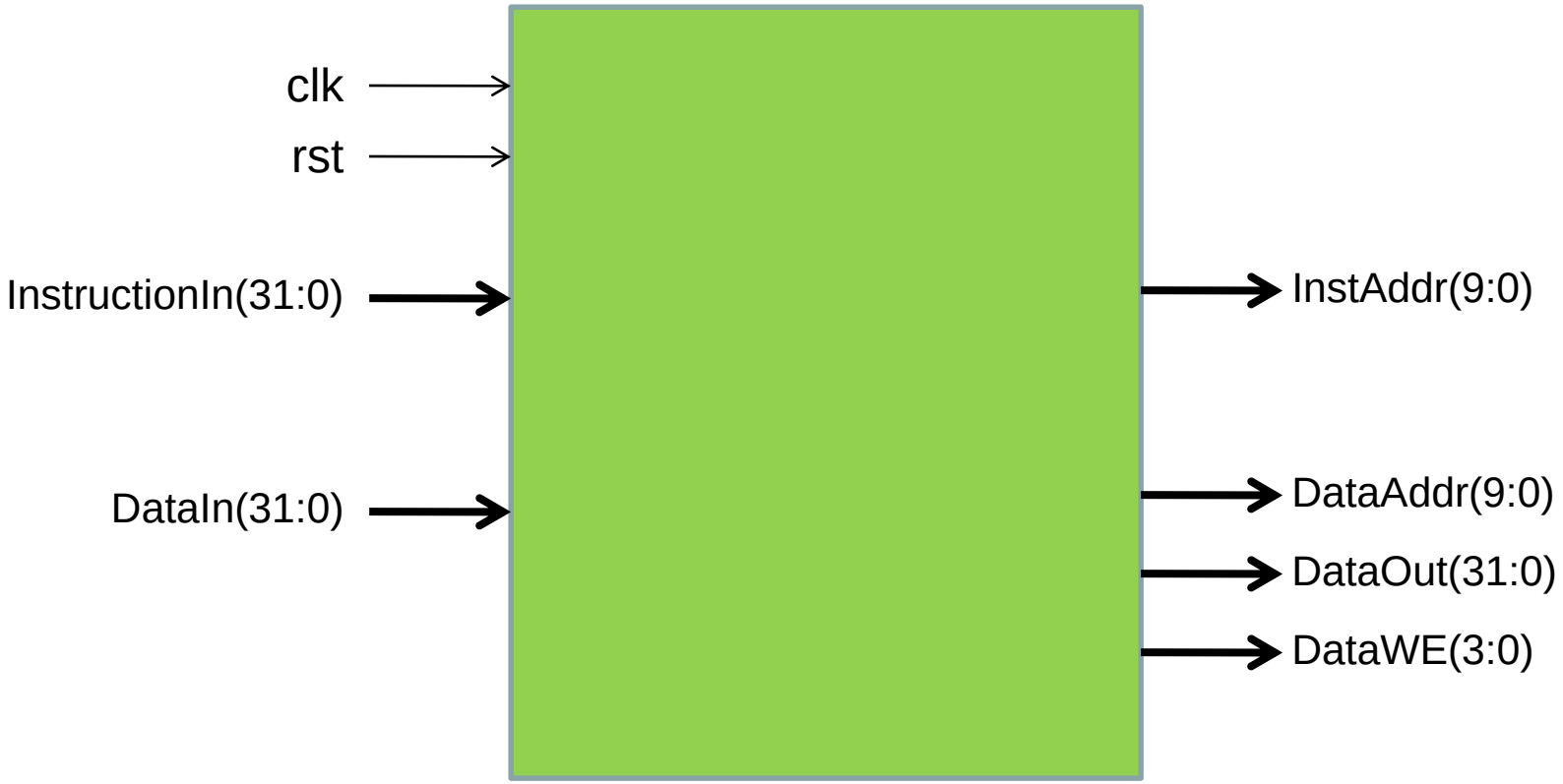
CPU Project

- Goal: design a pipelined processor that can execute 34 MIPS instructions on:

http://www.cse.sc.edu/~jbakos/611/mips/isa_detail.shtml

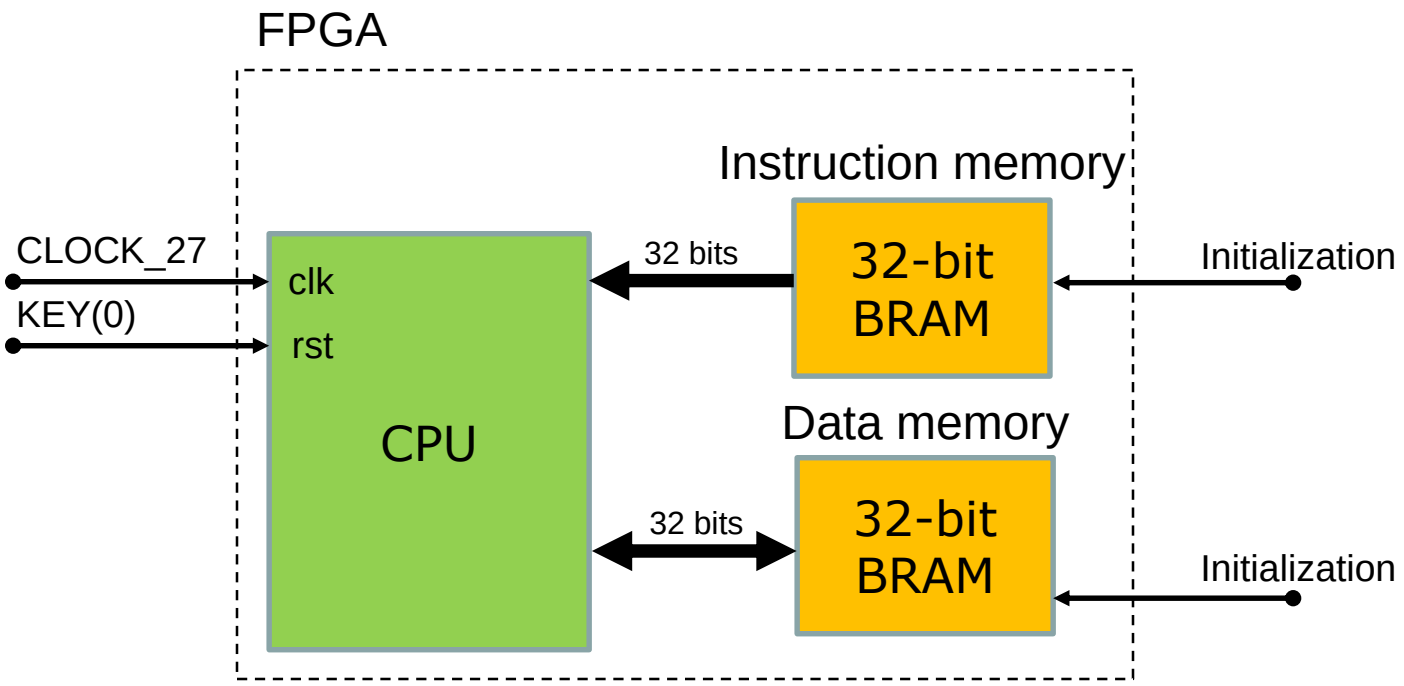
  - Use on-chip memory for program
    - Word addressed
  - Use on-chip memory for data
    - Word addressed but with byte-enables
  - Recall five steps:
    - fetch, decode, execute, memory, wb

CPU Module (Includes all Stages)



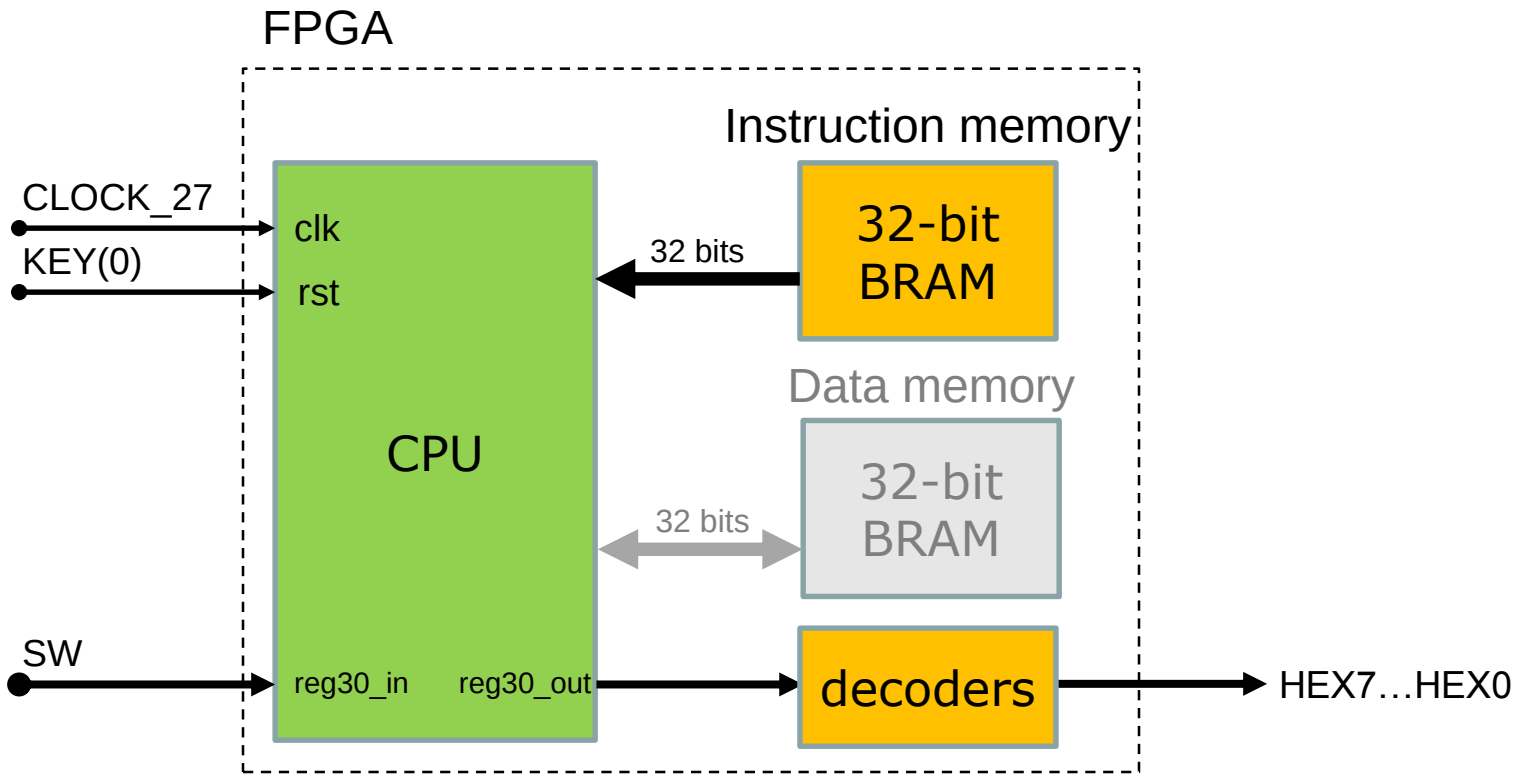
Top-Level Design

- Need two separate memories for instructions and data



Top-Level Design

- Need two separate memories for instructions and data
  - Each has different characteristics

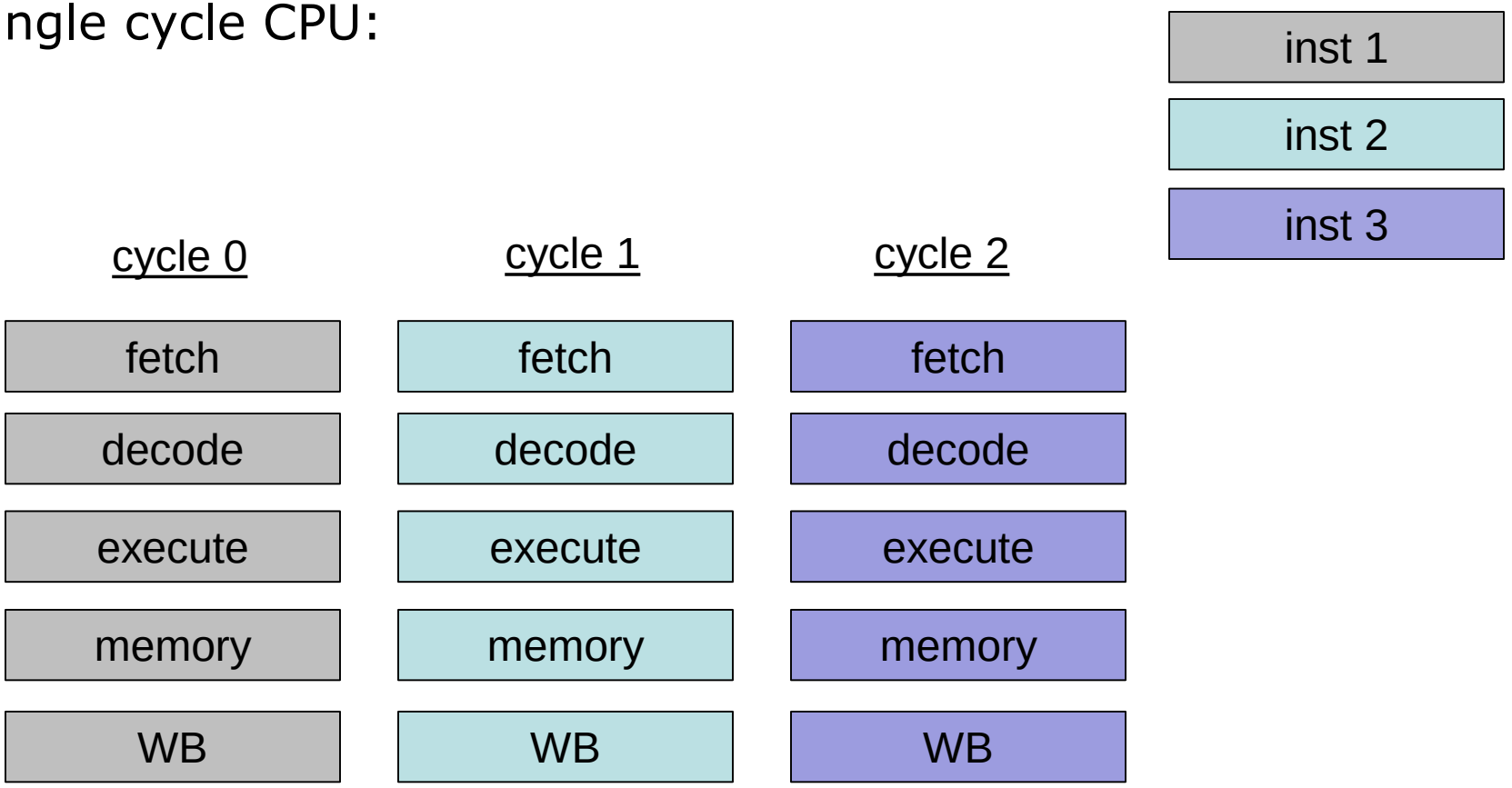


Execution Stages

- Basic steps: fetch, decode, execute, memory, write back
  - R-type computational instructions (ex. ADD $1, $2, $3):
    - fetch, decode, execute, write back
  - Branch instructions
    - fetch, decode, execute
  - Load instruction
    - fetch, decode, execute, memory, write back
  - Store instruction
    - fetch, decode, memory

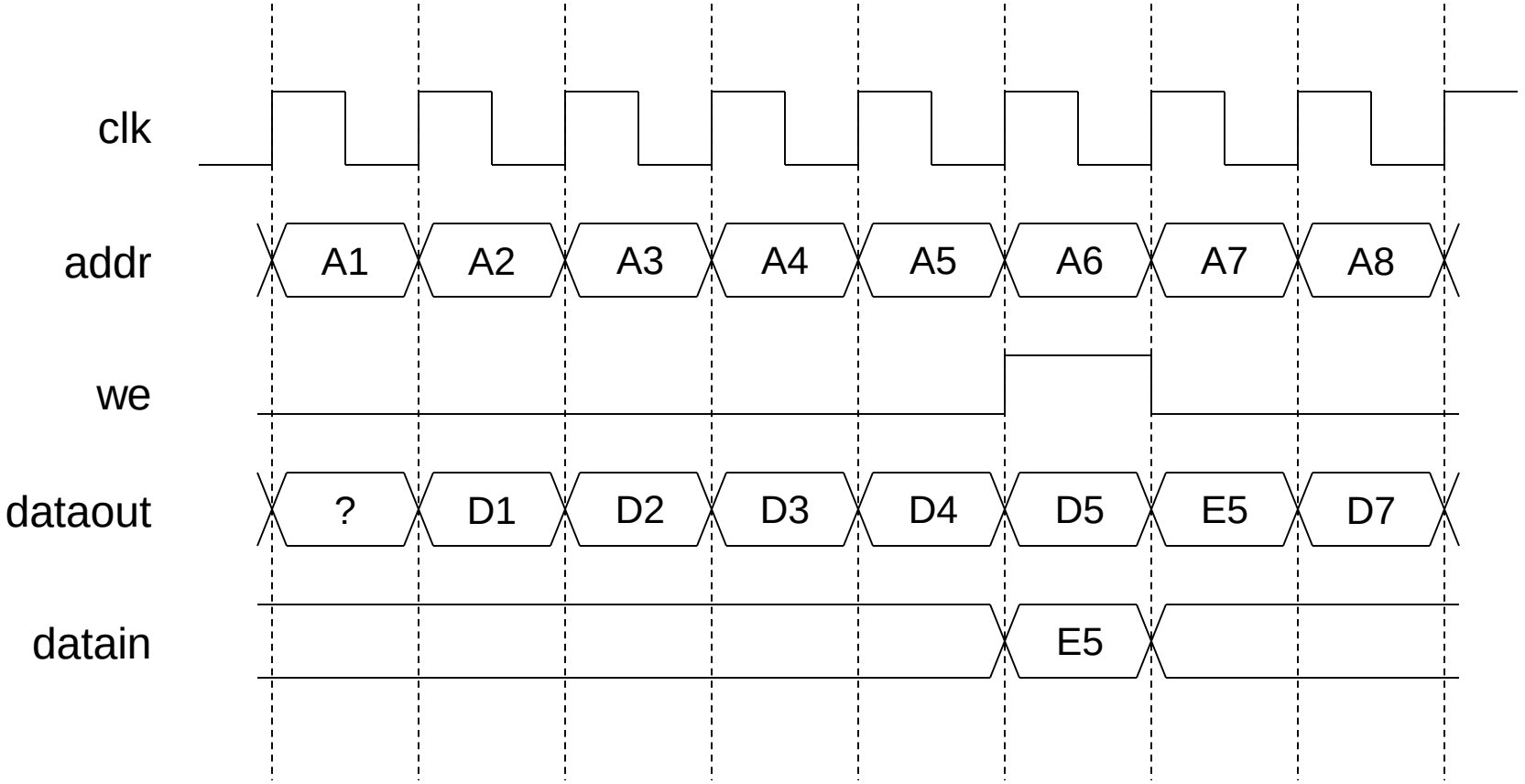
Execution Stages

- Single cycle CPU:

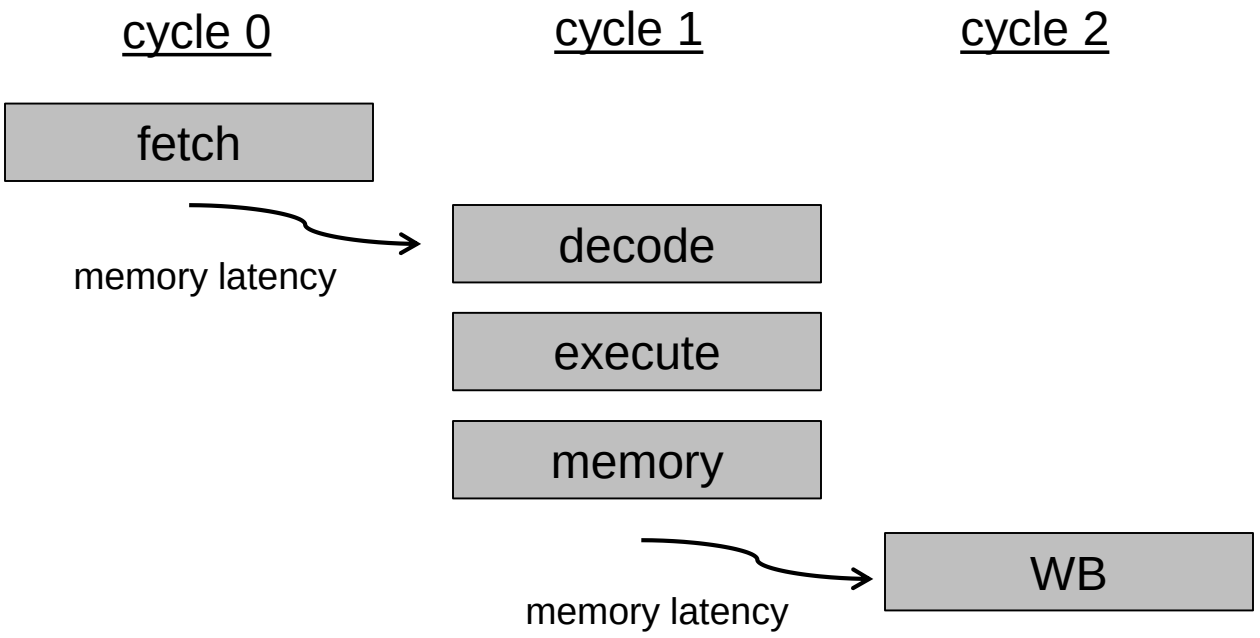


Synchronous Reads

- On-chip memory with synchronous reads

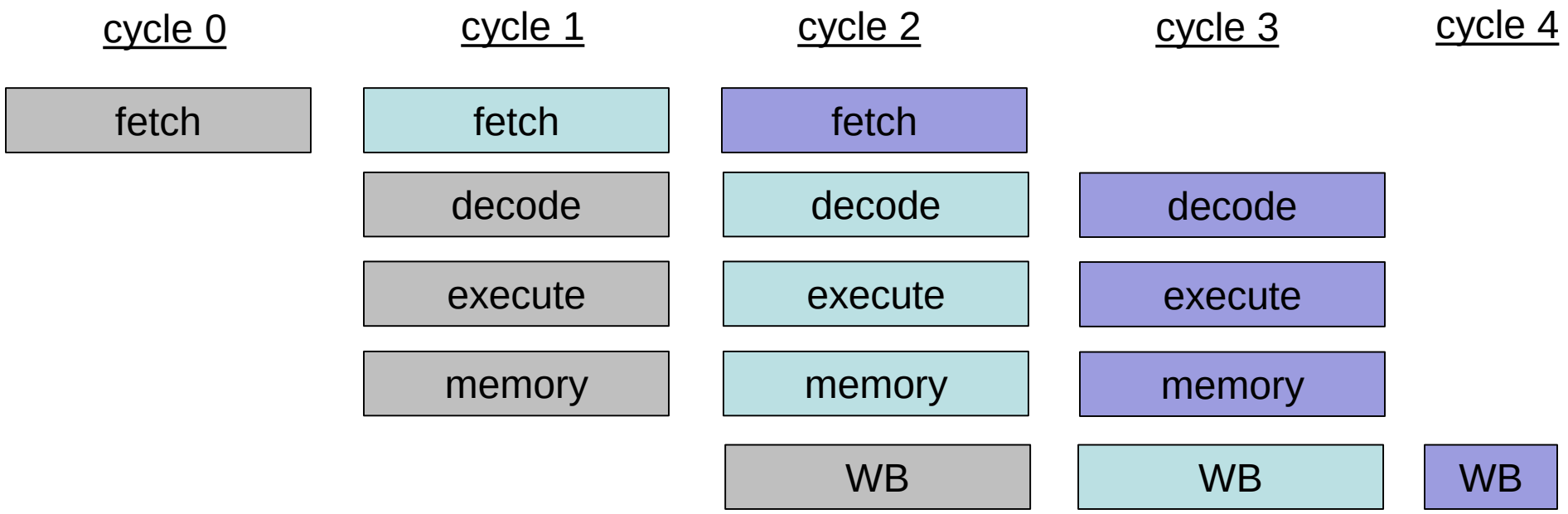


Execution Stages



Execution Stages

- Three stage pipeline



CPU Design

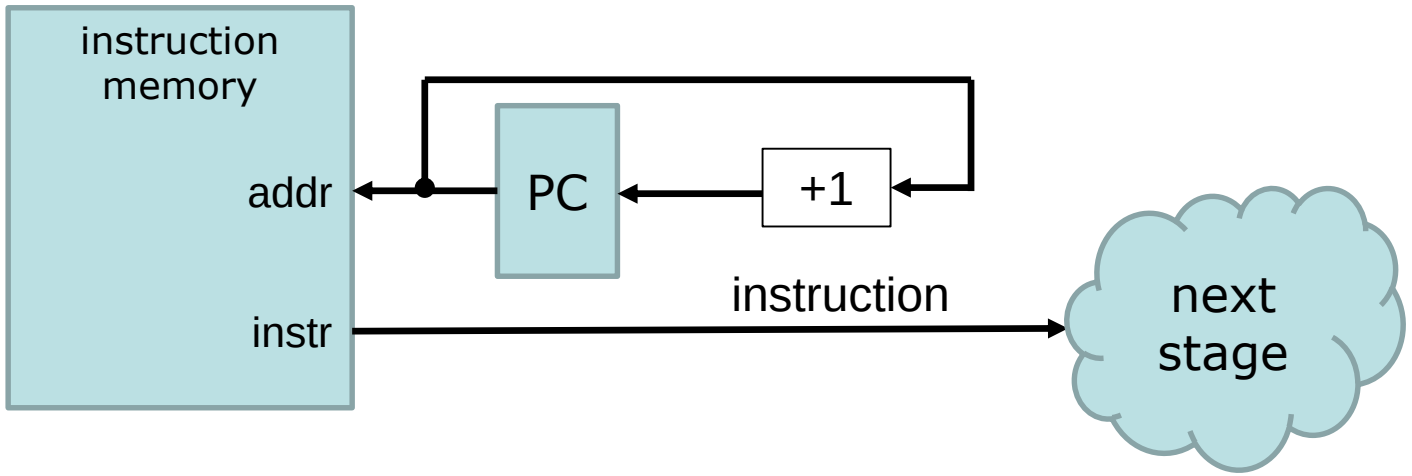
- Loads have a one-cycle latency
  - Need to separate MEM and WB
- Fetches have a one-cycle latency
  - Need to separate FETCH and DECODE
- Solution:
  - Three-stage pipeline:
    - FETCH, DECODE/EX/MEM, WB
    - F E W
  - Add pipeline registers between E and W
  - Loaded data is already delayed, so don't need a register for this
  - Leads to control hazard
    - Must flush FETCH for taken branches
    - Zero-out control signals in if branch is taken in previous cycle

Design Objectives

- Over the next four labs we will design a complete, single-issue pipelined MIPS processor core
  - **Lab 4**: Execute and WB stage supporting all R-type instructions except for JR and JALR
    - Test with a MIPS program that converts binary (switches) to decimal (HEX)
  - **Lab 5:** Support for load and store instructions
    - Test with a MIPS program that performs vector addition and summation
  - **Lab 6:** Support for branch and jump instructions
    - Test with a MIPS program that performs square root

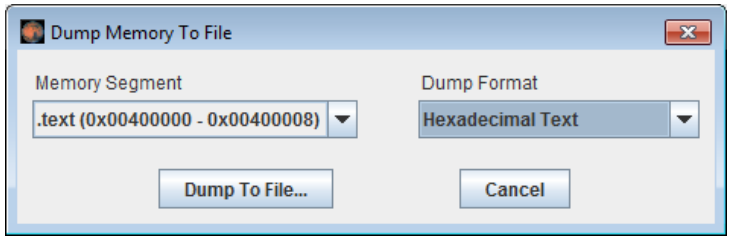
MIPS Fetch Stage

- Basic functionality with synchronous instruction memory



Initializing Instruction Memory

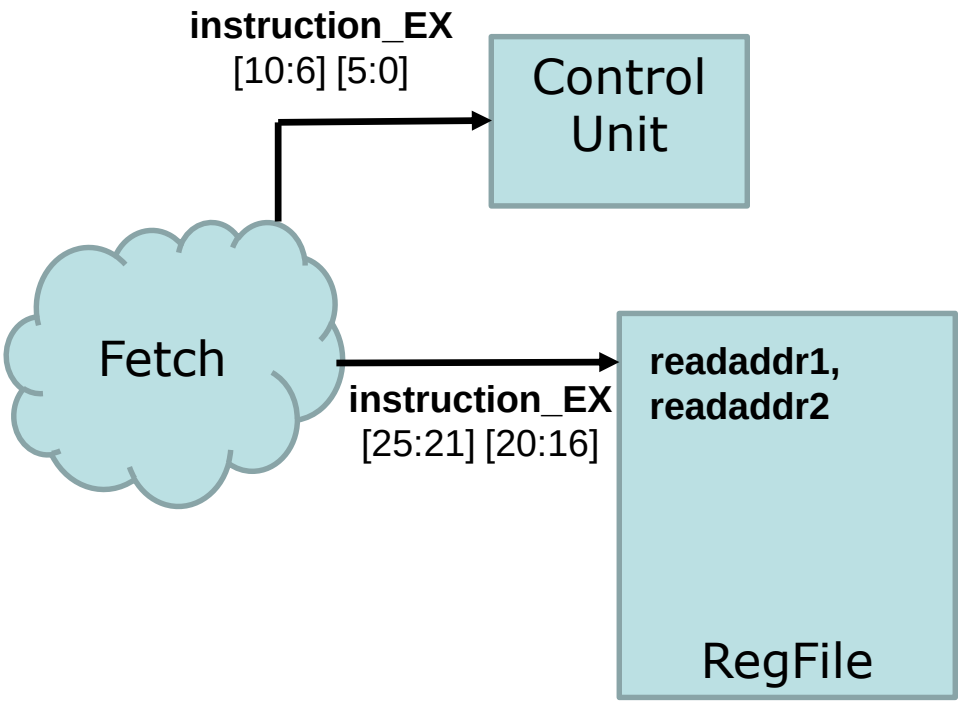
- Use script:
  - Use MARS to assemble (F3)
  - File | Dump Memory



  - Use this file to initialize instruction RAM:

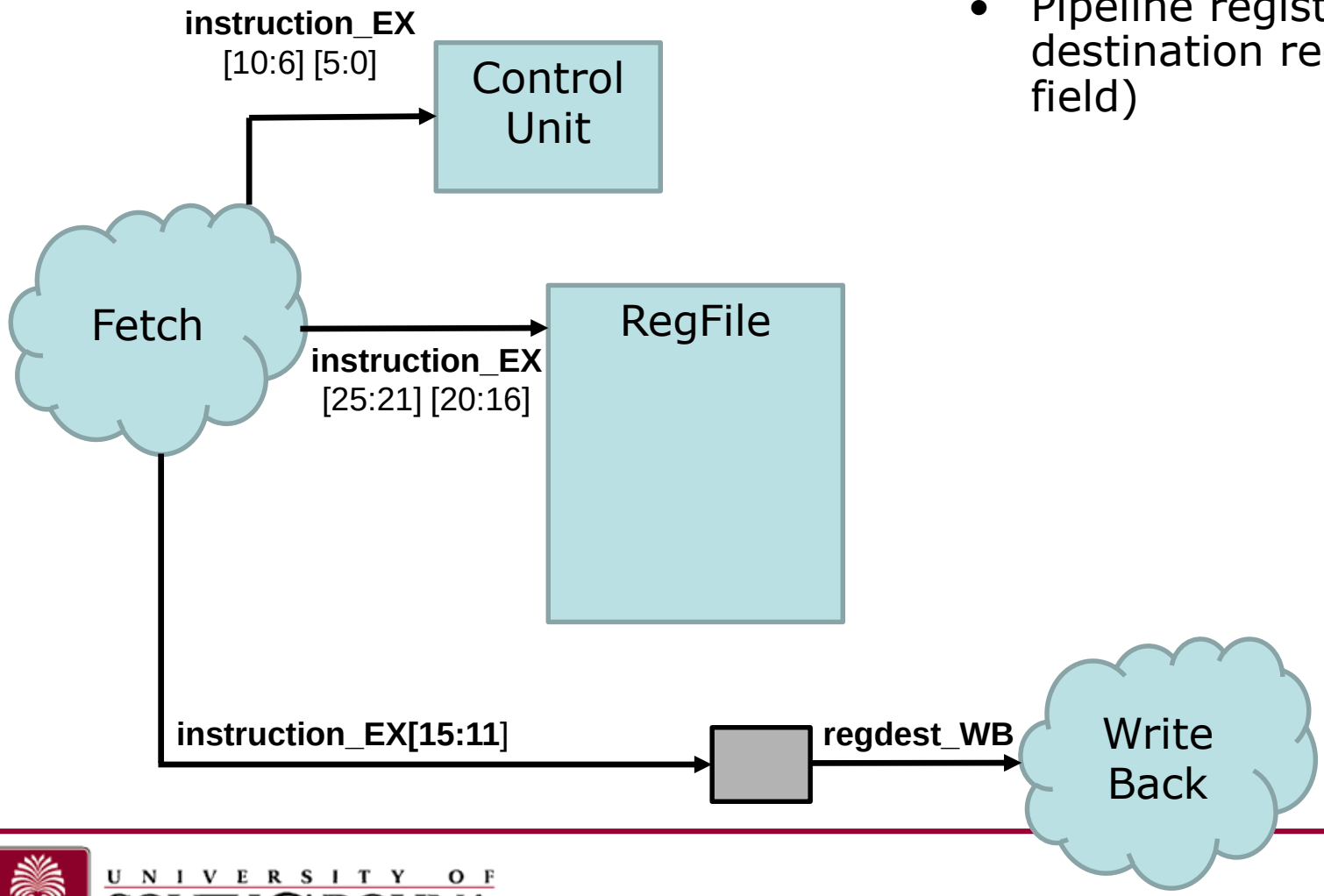
```
initial begin
  $readmemh("hexcode.txt", mem, 0, 1023);
end
```

EX Stage for R-type



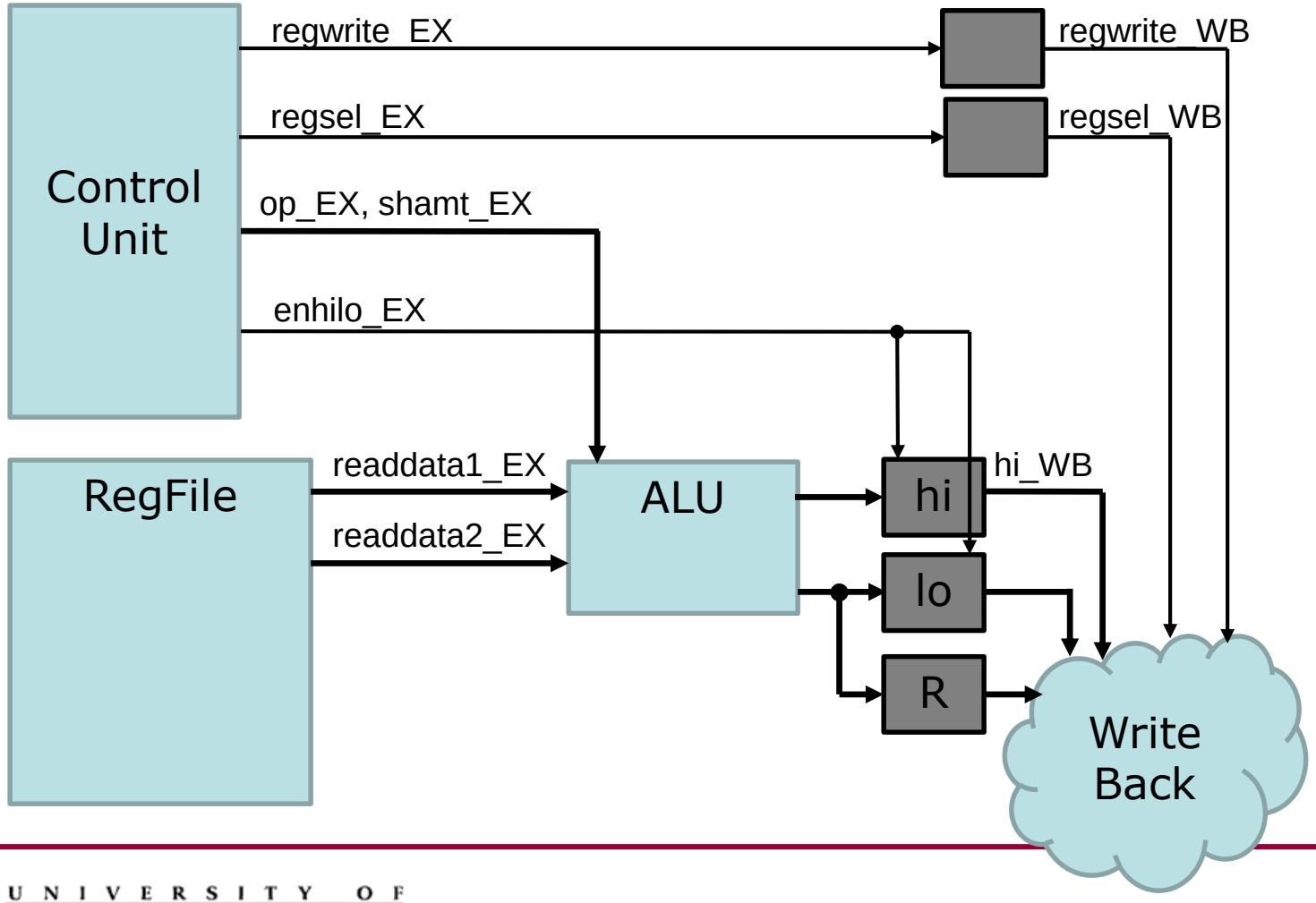
- Objective: fully implement instructions:
  - add, sub, addu, subu, and, or, nor, xor, slt, sltu
  - sll, srl, sra
  - mult, multu
  - mfhi, mflo
  - nop
- Instruction bits:
  - To control unit:
    - [31:26] opcode
    - [10:6] shamt
    - [5:0] function code
  - To register file:
    - [25:21] rs address (to readaddr1)
    - [20:16] rt address (to readaddr2)

EX Stage for R-type



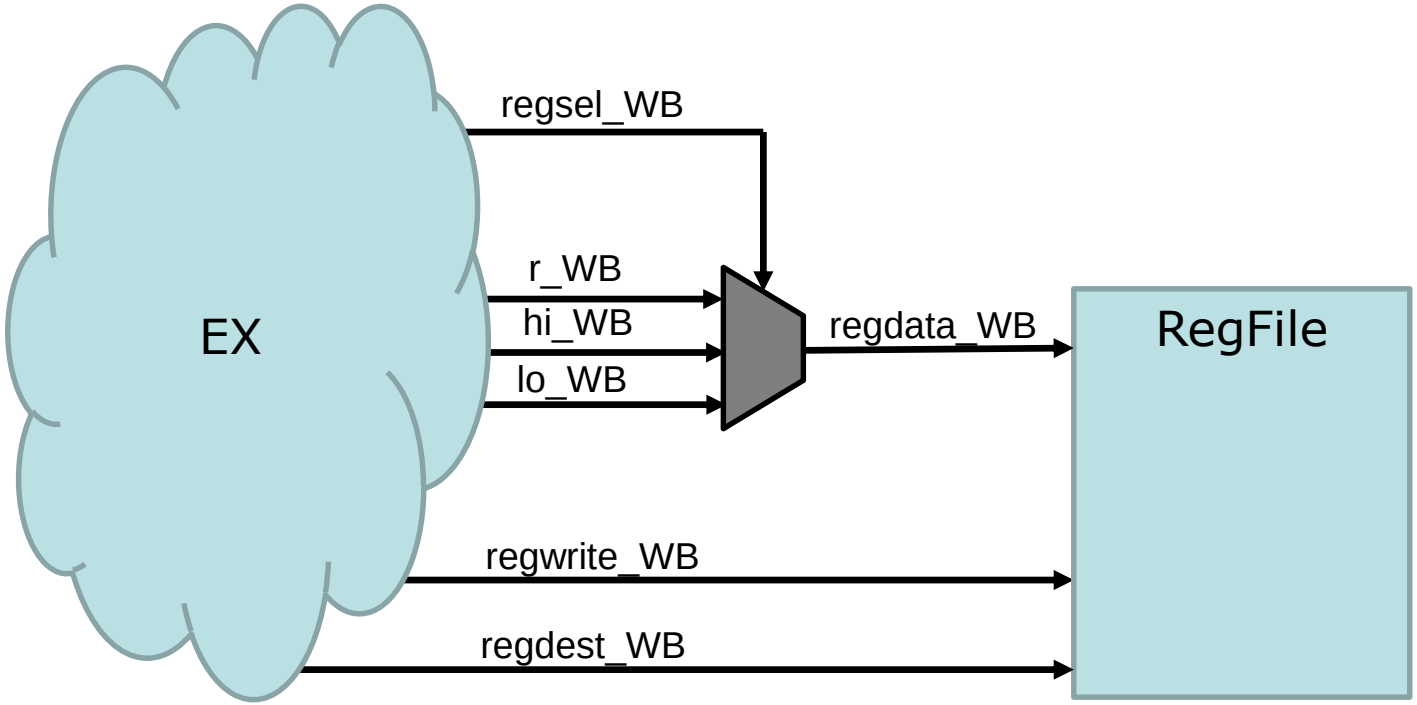
- Pipeline register for destination register (rd field)

EX Stage for R-type (Additional Signals)



WB Stage for R-type (Extra Signals)

- Basically just the write port on the register file



Control Unit for R-type

- Inputs (actual, in CPU design):
  - **instruction_EX[10:6]:** shift amount (used for sll, srl, sra instructions only)
  - **instruction_EX[5:0]:** function code (used for all R-type instructions)

- Outputs:
  - **op_EX[3:0]:** ALU operation (can use don't care X value for mflo, mfhi. nop)
  - **shamt_EX[4:0]:** set to instruction_EX[10:6] for sll, srl, sra, X for everything else
  - **enhilo_EX:** assert for mult/multu instructions
  - **regsel_EX[1:0]**: signal to the WB stage to indicate an ALU write on HI/LO write (1 for mfhi, 2 for mflo, 0 for everything else)
  - **regwrite_EX:** signal to the WB stage to write register (1 for all except MULT/MULTU)

Control Unit Implementation

```
module control_unit (input [5:0] function_code,…);

always @(*) begin
  if ((function_code == 6'b100000) | (function_code == 6'b100001)) begin // add, addu
    alu_op = 4'b0100;
    alu_shamt = 5'bX;
    enhilo = 1'b0;
    regsel = 2'b00;
    regwrite = 1'b1;
  end else if ((function_code == 6'b100010) | (function_code == 6'b100011)) begin // sub, subu
    alu_op = 4'b0101;
    alu_shamt = 5b'X;
    enhilo = 1'b0;
    regsel = 2'b00;
    regwrite = 1'b1;
  …
 end else if (function_code == 6'b011000) begin // mult (not multu)
   alu_op = 4'b0110;
    alu_shamt = 5b'X;
    enhilo = 1'b1;
    regsel = 2'b00;
    regwrite = 1'b0;  …
```

Example Test Program

```
.text
add $3, $1, $0          # ($3) <= 11
addu $4, $2, $0         # ($4) <= 17
sub $5, $0, $3          # ($5) <= -11
subu $6, $0, $4         # ($6) <= -17
mult $4, $6             # {hi,lo} <= -289
mflo $7                 # ($7) <= -289
mfhi $8                 # ($8) <= -1
multu $4, $6            # {hi,lo} <= 17 * 0xFFFFFFEF
mflo $9                 # ($9) <= -289
mfhi $10                # ($10) <= 16
and $11,$3,$4           # ($11) <= 1
or $12,$3,$4            # ($12) <= 27
xor $13,$3,$4           # ($13) <= 0x0000001A
nor $14,$3,$4           # ($14) <= 0xFFFFFFE4
nop                     # do nothing
sll $15,$3,1            # ($15) <= 22
srl $17,$5,1            # ($17) <= 0x7FFFFFFA
sra $19,$5,1            # ($19) <= -6
slt $21,$6,$4           # ($21) <= 1
sltu $22,$6,$4          # ($22) <= 0
```

Lab 4: R-type Instructions

- Objectives (due 10/27):
  1. Implement MIPS processor that implements instructions:
     `add, sub, addu, subu, mult, multu, and, or, nor, xor, sll, srl, sra, slt, sltu, mfhi, mflo, nop`

  2. Write two test programs:
     - test_program1.asm:
       - Tests all instructions (from previous slide)
       - Verify by viewing contents of register file in Modelsim
     - test_program2.asm:
       - Read switches as binary value (0 - 2^18-1)
       - Convert binary from switches to BCD on LEDs
     - Implement on DE2

Binary to Decimal Conversion

val	%10	/10
5678	8	567
567	7	56
56	6	5
5	5	0
0		

- reg=0
- while val /= 0
  - reg = reg << 4
  - reg = reg | val % 10
  - val = val / 10

- This will generate the digits in reverse, so you can connect the least significant BCD digit of the register to the most significant HEX

Work Around Divide

- Our CPU doesn't have a divide or a modulo instruction
- Since we're using a constant factor 10, we can use multiply

- Idea: use fixed-point multiply
  - Assume a decimal point at some location in a value:
  - Example (6,4)-fixed format:

1	0.	1	1	0	1

    - = 2 + 13/16 = 2.8125

  - Now assume we multiply a (32,0) value by a (32,32) value
  - Result would be (64,32) value
  - Use this to multiply a (32,0) value by 0.1
  - (32,32) representation of 0.1 = 2^32 / 10

Example

- Assume value = 234
  - Step 1: temp = 234 x 0.1 = 23.4
  - Step 2: temp2 = fractional(temp) = .4
  - Step 3: temp = whole(temp) = 23
  - Step 4: digit = temp2 x 10 = 4

  - Step 5: temp = temp x 0.1 = 2.3
  - Step 6: temp2 = fractional(temp) = .3
  - Step 7: temp = whole(temp) = 2
  - Step 8: digit = temp2 x 10 = 3

  - Step 9: temp = temp x 0.1 = 0.2
  - Step 10: temp2 = fractional(temp) = .2
  - Step 11: temp = whole(temp) = 0
  - Step 12: digit = temp2 x 10 = 2

  - Step 13: temp == 0 so finish

Example

- Fixed-point algorithm to convert base-2 integer (32,0) to base-10:
  - pt_one = 2^32/10 = 429496730
  - val <= value to convert
  - while val > 0
    - [hi,lo] = val * pt_one
    - val = hi // val is now floor(val/10)
    - [hi,lo] = lo * 10 // hi is now val % 10
    - digit = hi // hi is now floor base-10 LSD
- This algorithm generates the digits from LSD to MSD

Wrapper Design

- For testing on DE2:

```
module de2 (input CLOCK_50, input [3:0] KEY,
     output [6:0] HEX0, …, output [6:0] HEX7, input [17:0] SW);
// connect DE2 I/Os to CPU/mem/decoders
system my_system (.clk(CLOCK_50), .rst(~KEY[0]),
     .HEX0(HEX0), …, .HEX7(HEX7), .SW(SW));
```

- For testing with testbench:

```
module testbench ();
reg clk_tb;
reg rst_tb;
wire [6:0] HEX0;
…
initial begin rst_tb=1; #4; rst_tb=0; end
always begin clk_tb=0; #5; clk_tb=1; #5; end
system my_system (.clk(clk_tb), .rst(rst_tb),
     .HEX0(HEX0), …, .HEX7(HEX7), .SW(18'd15));
```

Register Initialization

- In register file, add:

```
initial begin
  reg[1] <= 32'd <value for 10>
  reg[2] <= 32'd <value for 0.1>
end
```