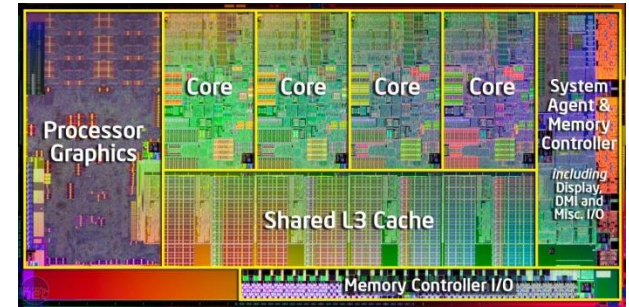




고려대학교
KOREA UNIVERSITY



COSE222, COMP212 Computer Architecture

Lecture 5. MIPS Processor Design

Single-cycle MIPS #1

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

Introduction

- Microarchitecture means a lower-level hardware structure that is able to execute instructions
- Multiple implementations for a single ISA
 - **Single-cycle**
 - Each instruction is executed in a single cycle
 - It suffers from the long critical path delay, limiting the clock frequency
 - **Multi-cycle**
 - Each instruction is broken up into a series of shorter steps
 - Different instructions use **different** numbers of steps, so simpler instructions completes faster than more complex ones
 - **Pipeline (5 stage)**
 - Execution is broken up into a series of steps
 - All the instructions use the **same** number of steps
 - Multiple instructions (up to 5) are executed **simultaneously**

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

Revisiting Performance

CPU Time = # insts X CPI X clock cycle time (T) = # insts X CPI / f

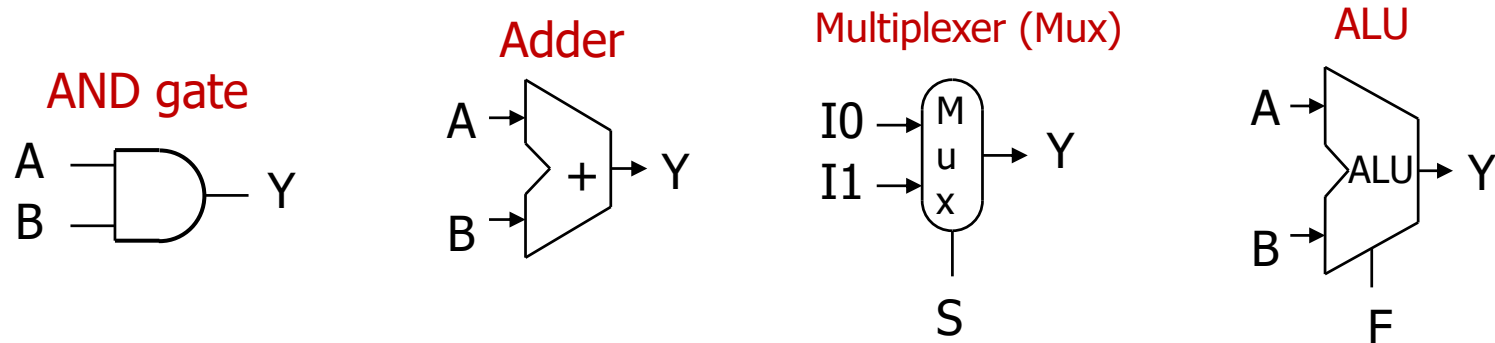
$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
 - **Algorithm** affects the instruction count
 - **Programming language** affects the instruction count and CPI
 - **Compiler** affects the instruction count and CPI
 - **Instruction set architecture** affects the instruction count, CPI, and T (f)
 - **Microarchitecture** (Hardware implementation) affect CPI and T (f)
 - **Semiconductor technology** affects T (f)
- Challenges in designing microarchitecture is to satisfy constraints of **cost**, **power** and **performance**

Revisiting Logic Design Basic

- Combinational logic

- Output is directly determined by current input

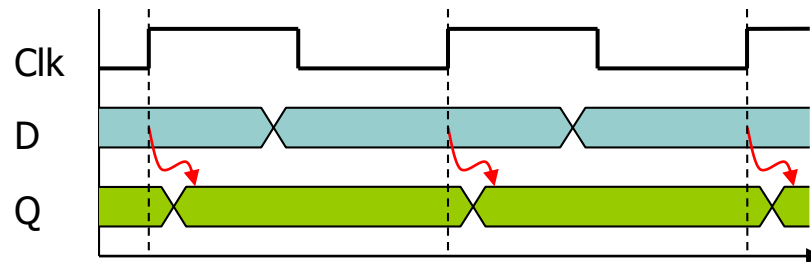
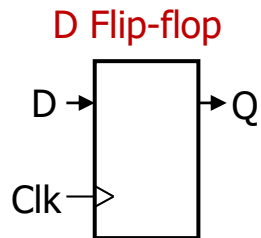


- Sequential logic

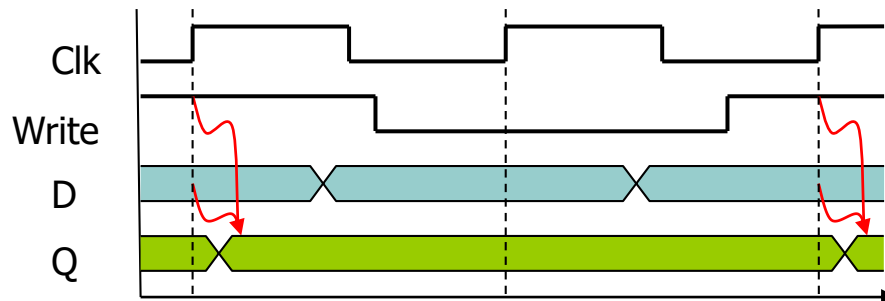
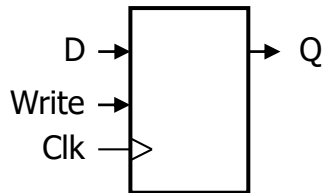
- Output is determined not only by current input, but also internal state (i.e., previous inputs)
- Sequential logic needs state elements to store information
 - Flip-flops** and **latches** are used to store the state information. But, avoid using latch in digital design

Revisiting State Element

- Registers (implemented with flip-flops) store data in a circuit
 - Clock signal determines **when** to update the stored value
 - Rising-edge triggered: update when clock changes from 0 to 1
 - Falling-edge triggered: update when clock changes from 1 to 0
 - Data input determines **what (0 or 1)** to update to the output

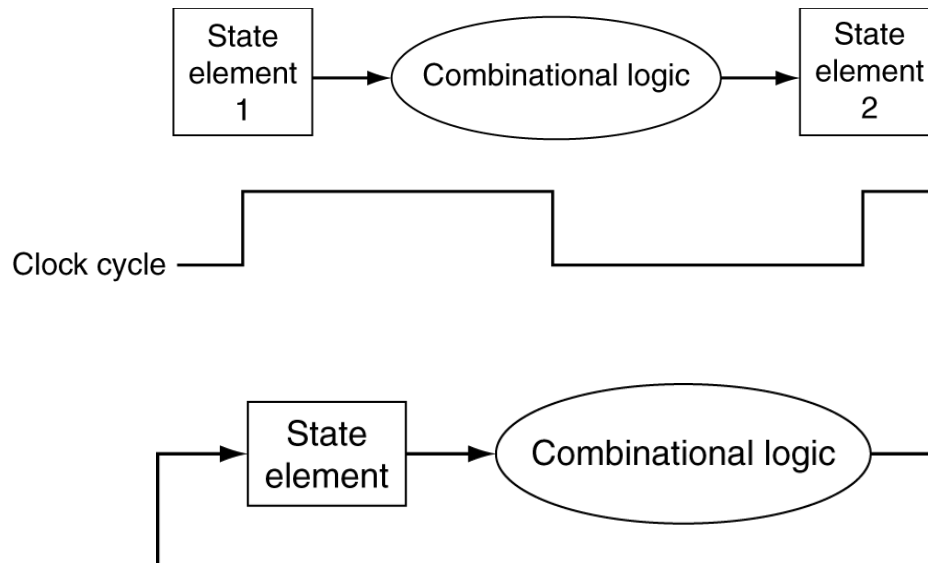


- Register with write control (or enable)
 - Only updates on clock edge **when write control (enable) input is 1**



Clocking Methodology

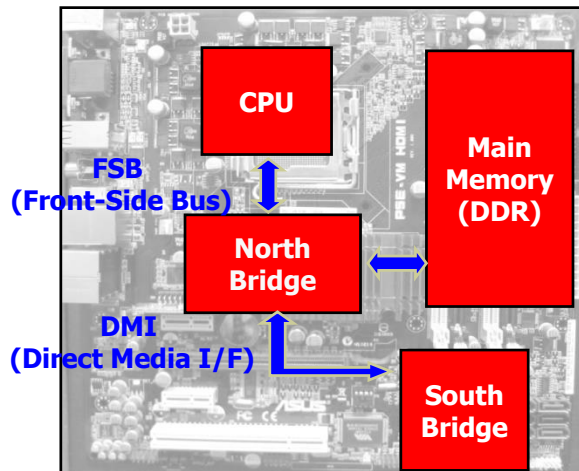
- Virtually all digital systems are synchronous to the clock
- Combinational logic sits between state elements (flip-flops)
- Combinational logic produces its intended data during clock cycles
 - Input from state elements
 - Output to the next state elements
 - Longest delay determines the clock period (frequency)



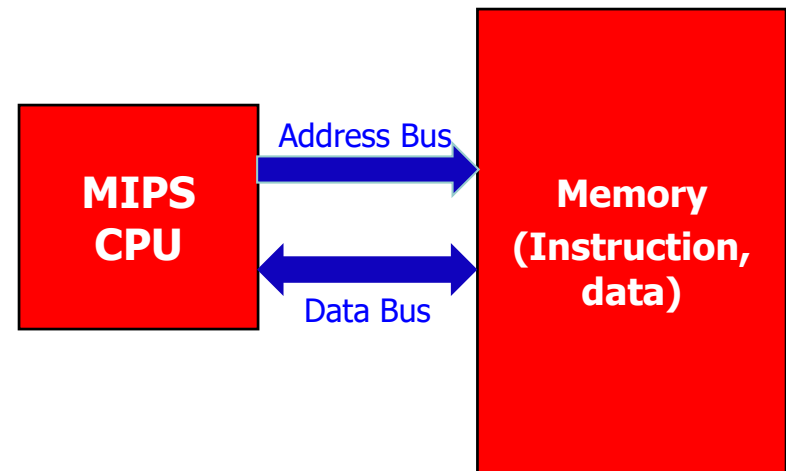
Overview

- We are going to design a MIPS CPU that is able to execute the machine code we discussed so far
- For the sake of your understanding, we simplify the CPU and its system structure

Real-PC system

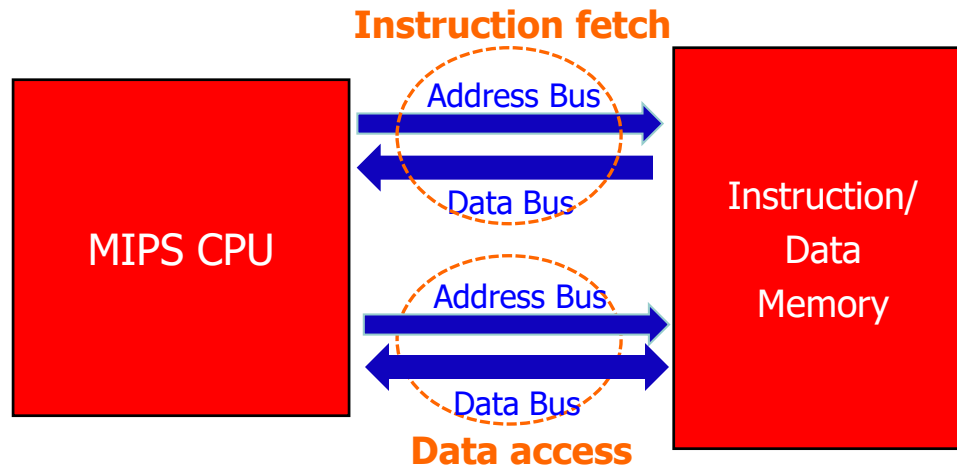


Simplified



Our MIPS Model

- Our MIPS CPU model has separate connections to memory
 - Actually, this structure is more realistic as we will see when we study caches

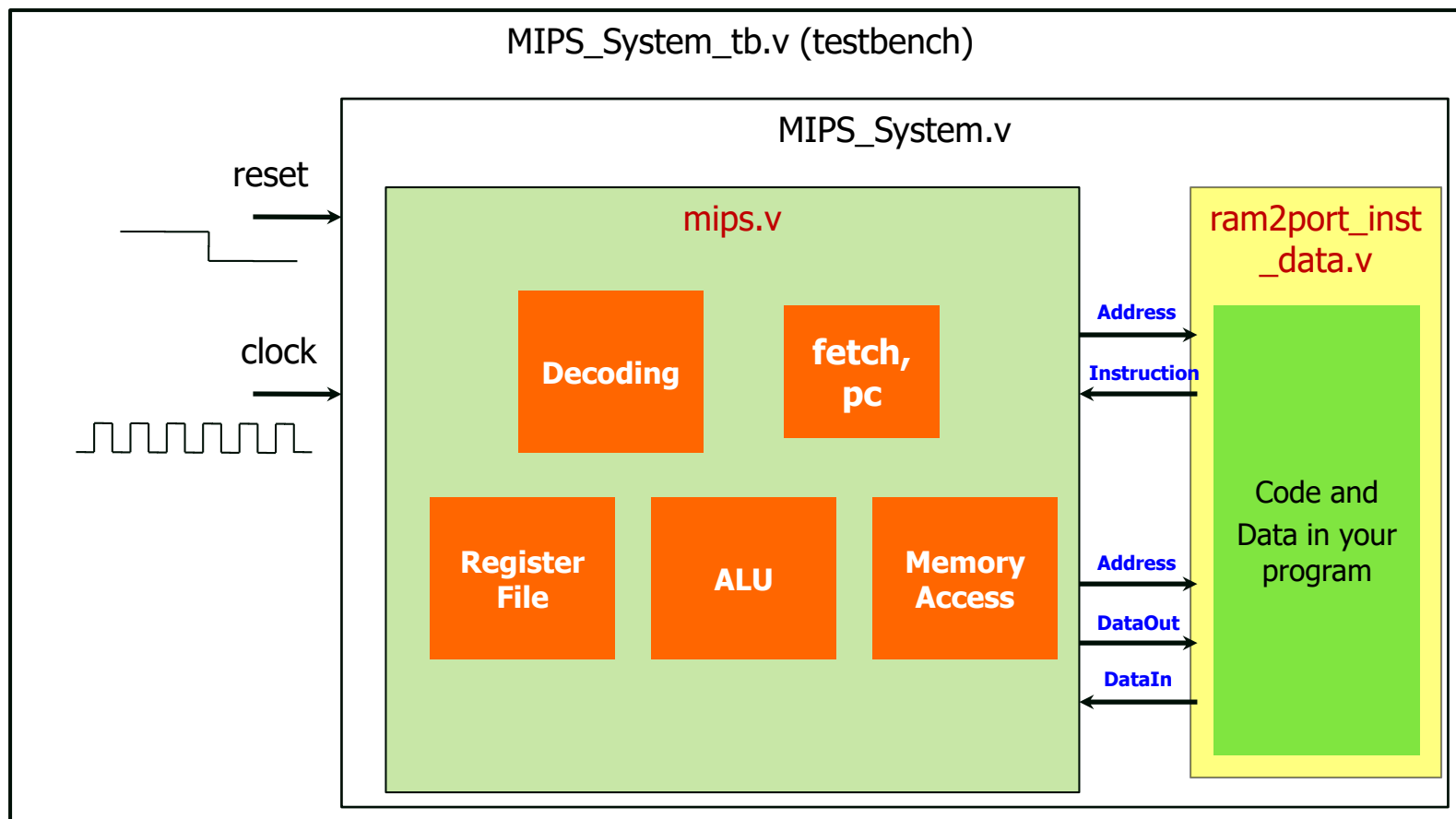


- We use both structural and behavioral modeling with Verilog-HDL
 - Behavioral modeling descriptively specifies what a module does
 - For example, the lowest modules (such as ALU and register files) are designed with the behavioral modeling
 - Structural modeling describes a module from simpler modules via instantiations
 - For example, the top module (such as `mips.v`) are designed with the structural modeling

Overview

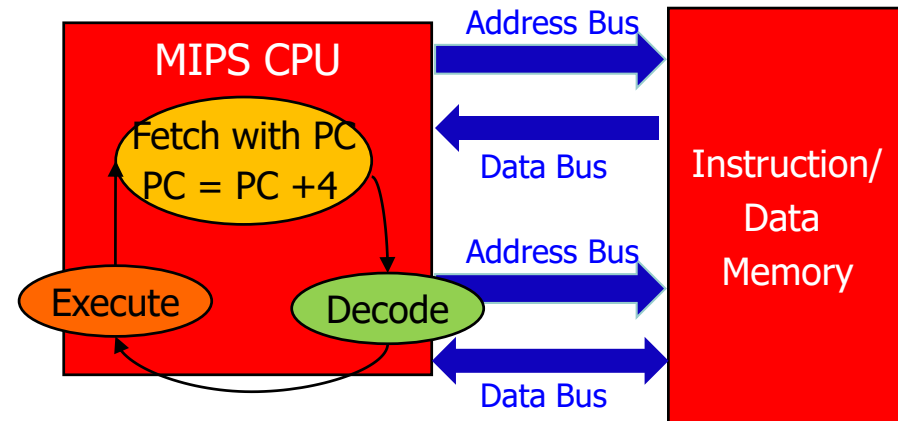
- Microarchitecture is composed of **datapath** and **control**
 - **Datapath** operates on words of data
 - Datapath elements are used to operate on or hold data within a processor
 - In MIPS implementation, datapath elements include the register file, ALU, muxes, and memory
 - **Control** tells the datapath how to execute instructions
 - Control unit receives the current instruction from memory and tells the datapath how to execute that instruction
 - Specifically, the control unit produces mux select, register enable, ALU control, and memory write signals to control the operation of the datapath
- Our MIPS implementation is simplified by designing only
 - Data processing instructions: **add, sub, and, or, slt**
 - Memory access instructions: **lw, sw**
 - Branch instructions: **beq, j**

Overview of Our Design

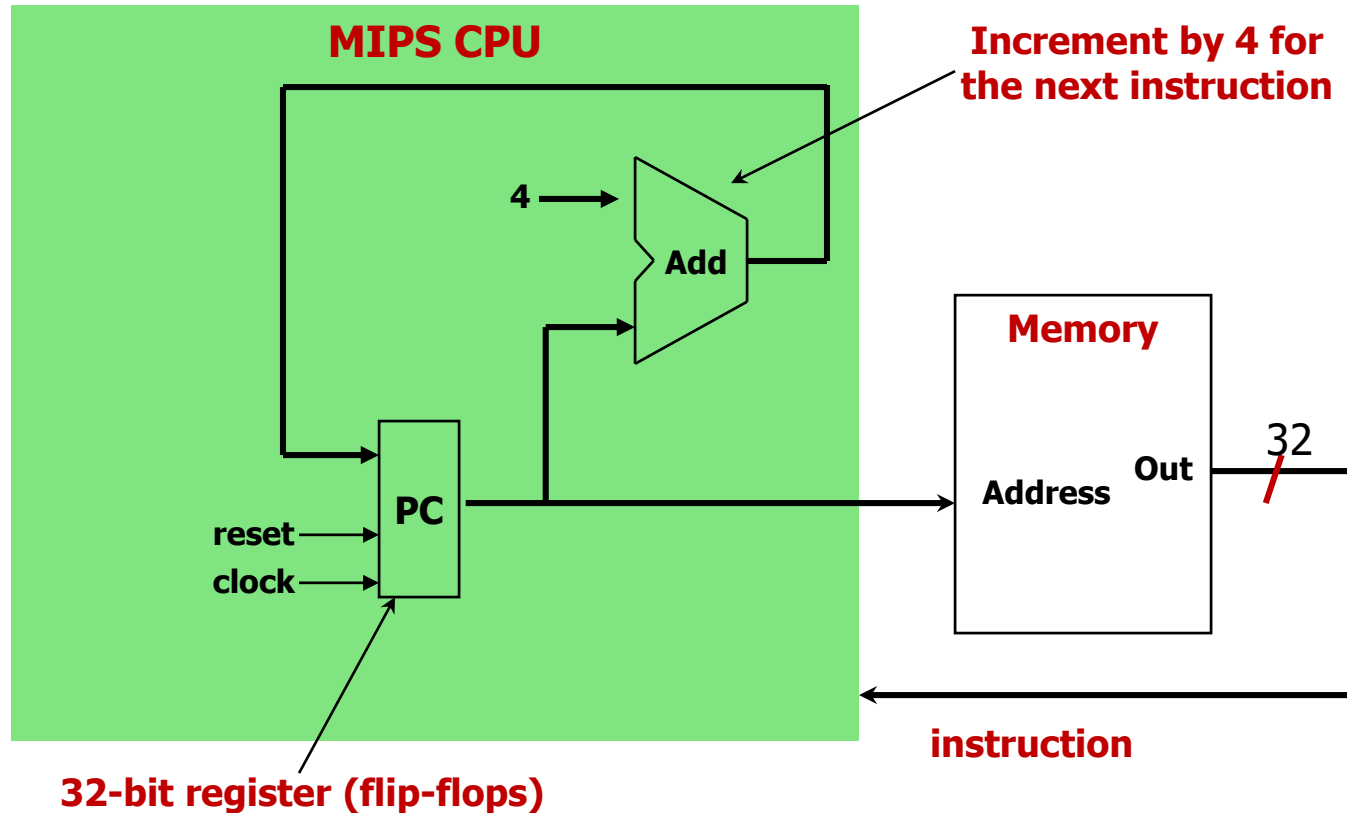


Instruction Execution in CPU

- Generic steps of the instruction execution in CPU
 - **Fetch** uses the program counter (PC) to supply the instruction address and fetch instruction from memory
 - Extract opcode: determine what operation should be done
 - Extract operands: register numbers or immediate from fetched instruction
 - **Decoding** decodes instruction and reads operands
 - Use ALU to calculate (depending on instruction class)
 - Arithmetic or logical result
 - Memory address for load/store
 - Branch target address
 - Access memory for load/store
 - **Next Fetch**
 - $PC \leftarrow \text{target address or } PC + 4$

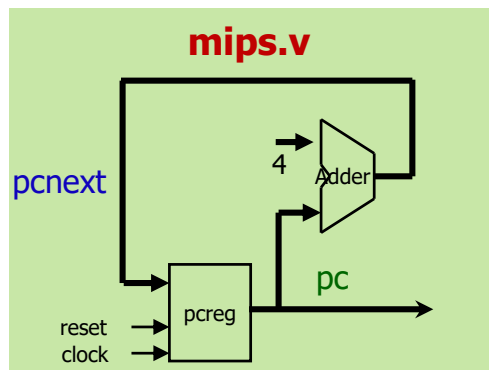


Instruction Fetch



- What is PC on reset?
 - MIPS initializes PC to 0xBFC0_0000
 - For the sake of simplicity, let's initialize the PC to 0x0000_0000 in our design

Instruction Fetch Verilog Model



```
module pcreg (
    input clk,
    input reset,
    output reg [31:0] pc,
    input [31:0] pcnext);

always @(posedge clk, posedge reset)
begin
    if (reset) pc <= 32'h00000000;
    else pc <= pcnext;
end

endmodule
```

```
module adder(
    input [31:0] a,
    input [31:0] b,
    output [31:0] y);

    assign y = a + b;

endmodule
```

```
module mips(
    input clk,
    input reset,
    output [31:0] pc,
    input [31:0] instr);

wire [31:0] pcnext;

// instantiate pc
pcreg mips_pc (.clk (clk),
               .reset (reset),
               .pc (pc),
               .pcnext (pcnext));

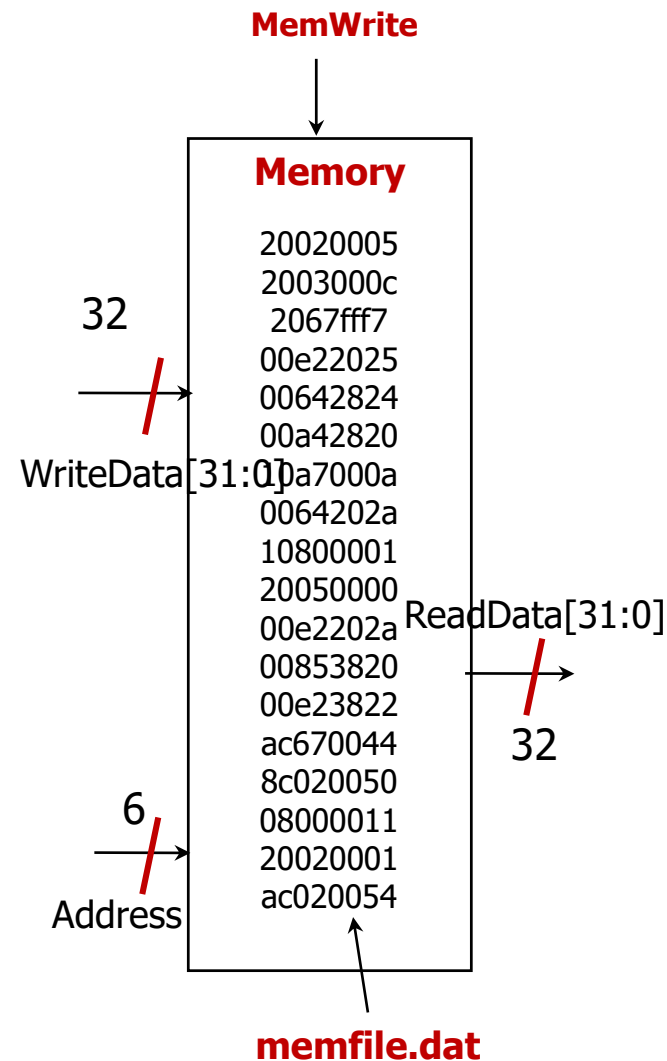
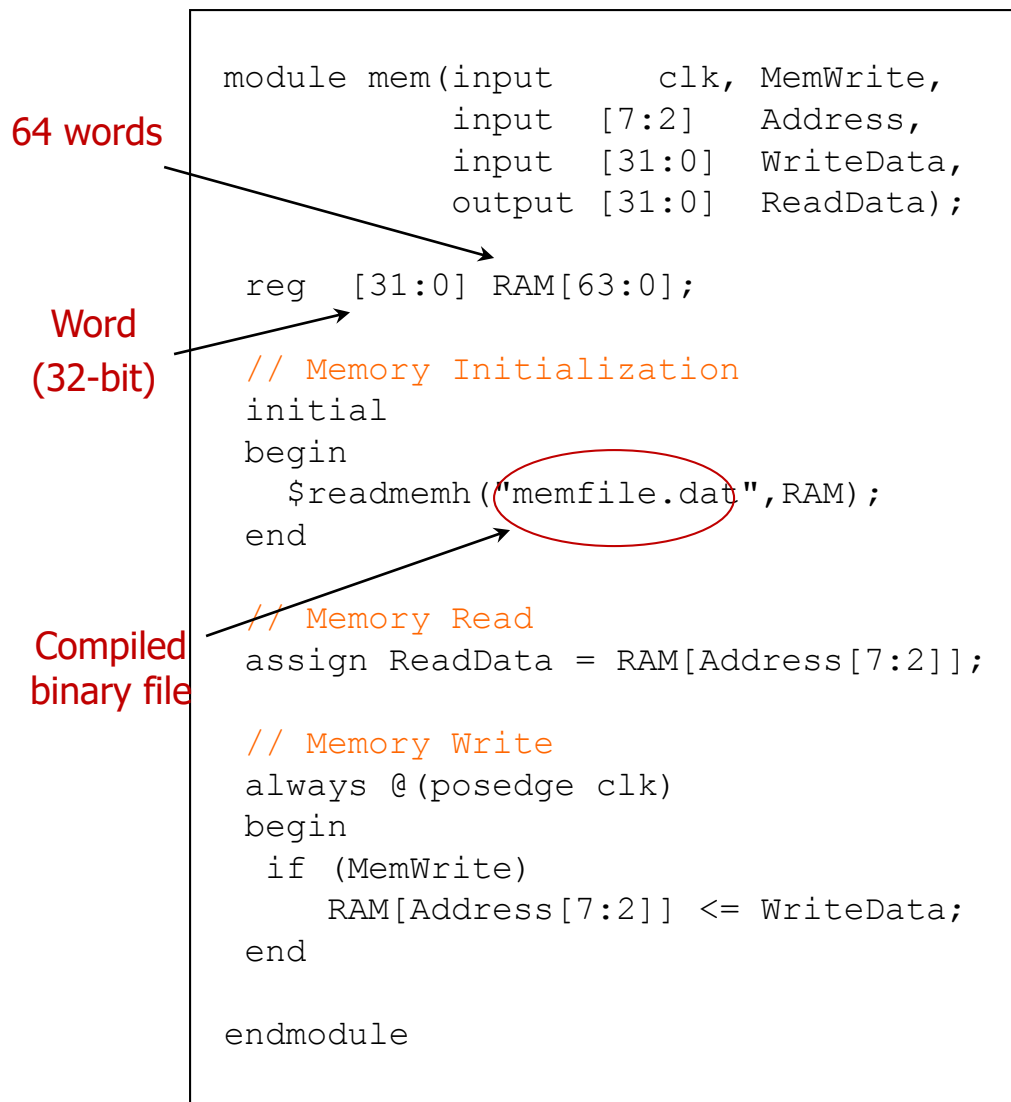
// instantiate adder
adder pcadd4 (.a (pc),
              .b (32'b100),
              .y (pcnext));

endmodule
```

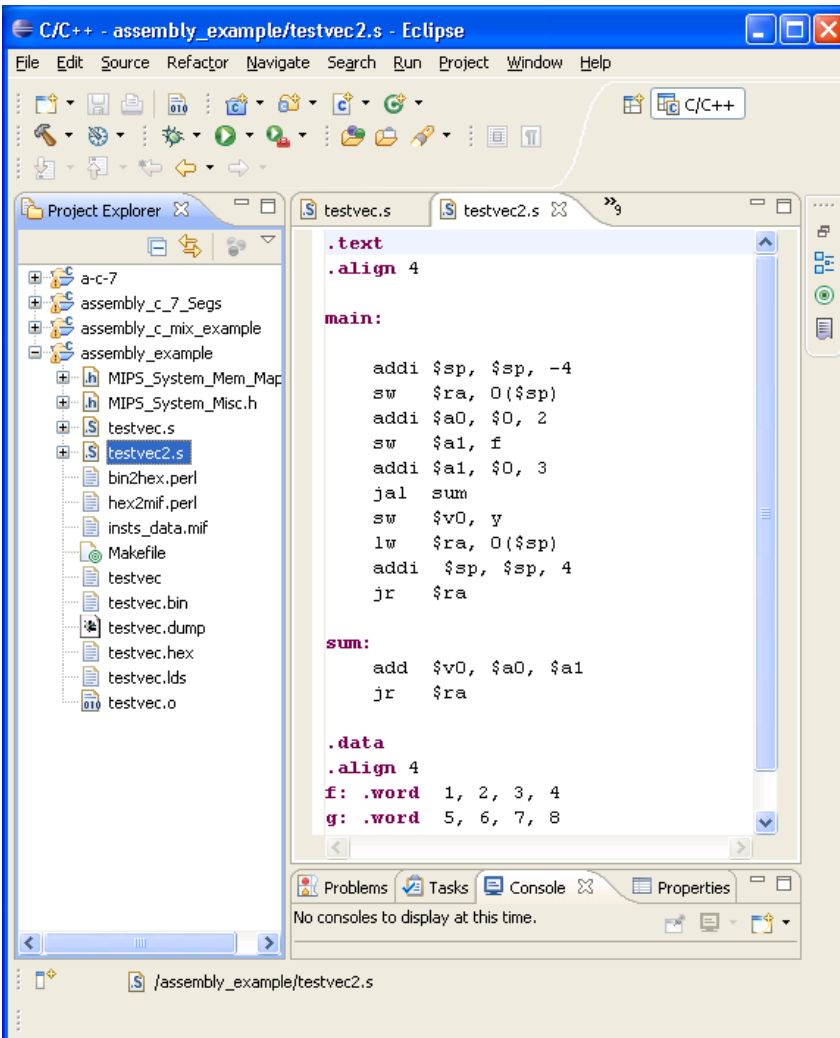
Memory

- As studied in the Computer Logic Design, memory is classified into RAM (Random Access Memory) and ROM (Read-Only Memory)
 - RAM is classified into DRAM (Dynamic RAM) and SRAM (Static RAM)
 - DDR is a kind of DRAM
 - DDR is a short form of DDR (Double Data Rate) SDRAM (Synchronous DRAM)
 - DDR is used as main memory in modern computers
- We use a Cyclone-II (Altera FPGA)-specific memory model because we port our design to the Cyclone-II FPGA

Generic Memory Model in Verilog



Simple MIPS Test Code



The screenshot shows the Eclipse IDE with the file `testvec2.s` open. The code is as follows:

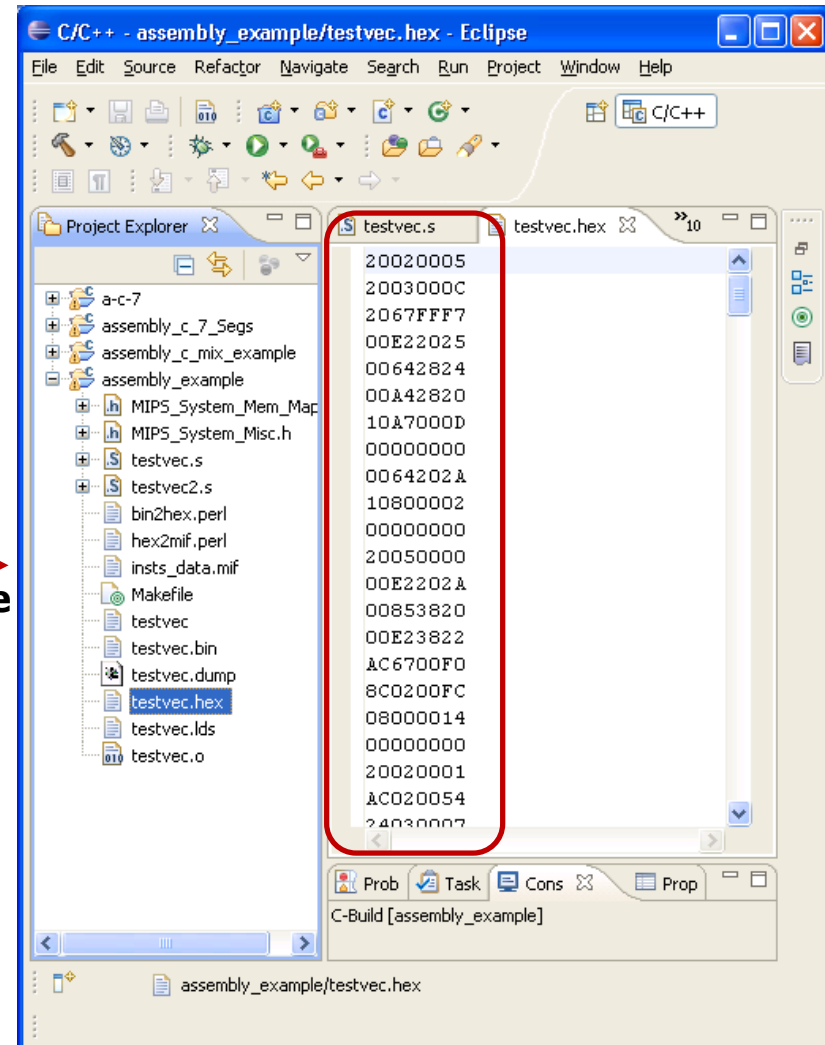
```
.text
.align 4

main:
    addi $sp, $sp, -4
    sw   $ra, 0($sp)
    addi $a0, $0, 2
    sw   $a1, f
    addi $a1, $0, 3
    jal  sum
    sw   $v0, y
    lw   $ra, 0($sp)
    addi $sp, $sp, 4
    jr   $ra

sum:
    add  $v0, $a0, $a1
    jr   $ra

.data
.align 4
f: .word 1, 2, 3, 4
g: .word 5, 6, 7, 8
```

→
assemble

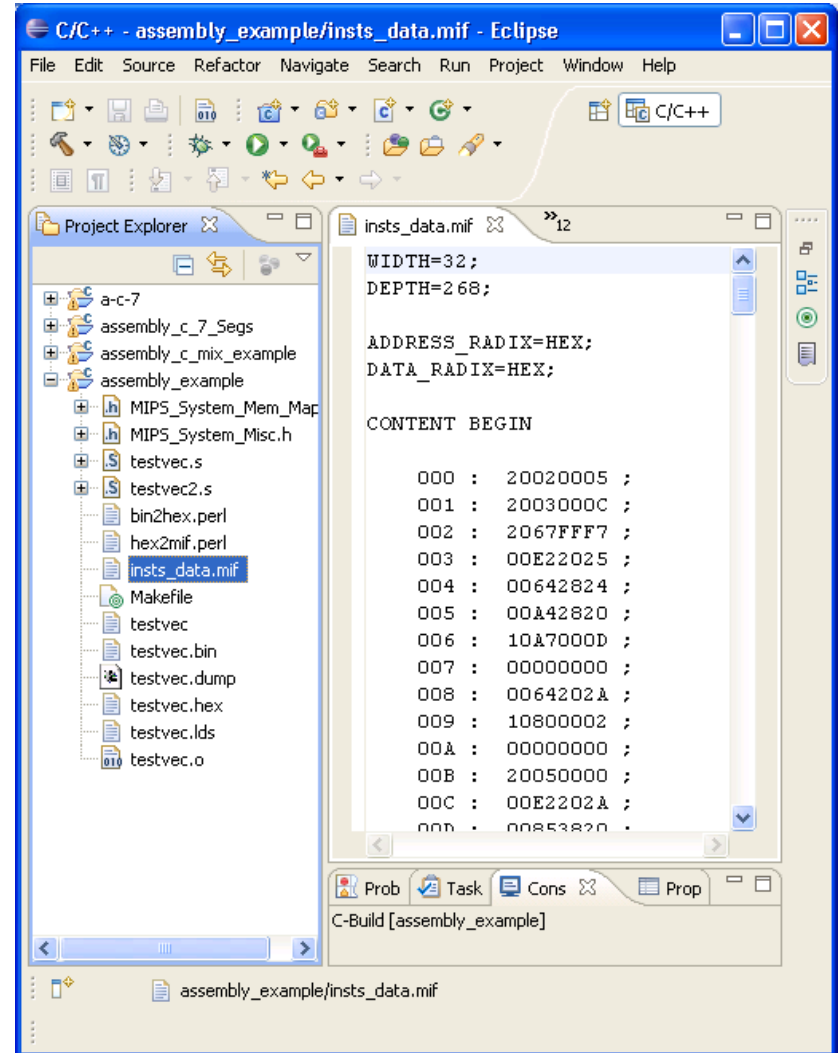


The screenshot shows the Eclipse IDE with the file `testvec.hex` open. The output is a list of hexadecimal values, with the first 16 values highlighted by a red box:

```
20020005
2003000C
2067FFF7
00E22025
00642824
00A42820
10A7000D
00000000
0064202A
10800002
00000000
hex2mif.perl
insts_data.mif
20050000
00E2202A
00853820
00E23822
AC6700F0
8C0200FC
08000014
00000000
20020001
AC020054
24030007
```

Our Memory

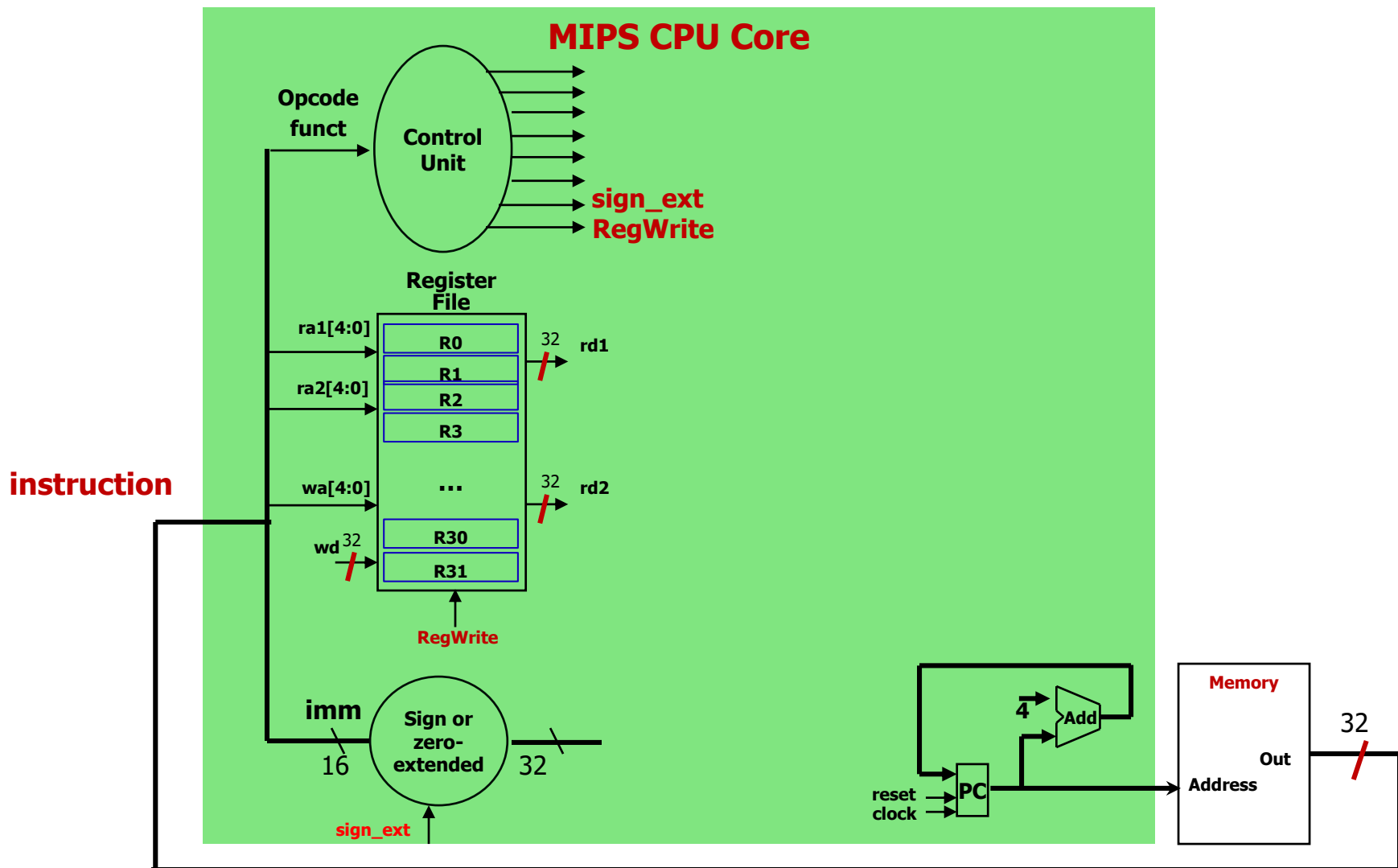
- As mentioned, we use a Cyclone-II (Altera FPGA)-specific memory model because we port our design to the Cyclone-II FPGA
 - Prof. Suh has created a memory model using MegaWizard in Quartus-II
 - To initialize the memory, it requires a special format called `mif`
 - Prof. Suh wrote a perl script to generate the mif-format file
 - Check out `Makefile`
 - For synthesis and simulation, just copy `insts_data.mif` to `MIPS_System_Syn` and `MIPS_System_Sim` directories



Instruction Decoding

- Instruction decoding separates the fetched instruction into the fields according to the instruction types (R, I, and J types)
 - **Opcode** and **funct** fields determine which operation the instruction wants to do
 - Control logic should be designed to supply control signals to datapath elements (such as ALU and register file)
 - **Operands**
 - Register numbers in the instruction are sent to the register file
 - Immediate field is either sign-extended or zero-extended depending on instructions

Schematic with Instruction Decoding



Register File in Verilog

```
module regfile(input      clk,
               input      RegWrite,
               input  [4:0] ra1, ra2, wa,
               input  [31:0] wd,
               output [31:0] rd1, rd2);

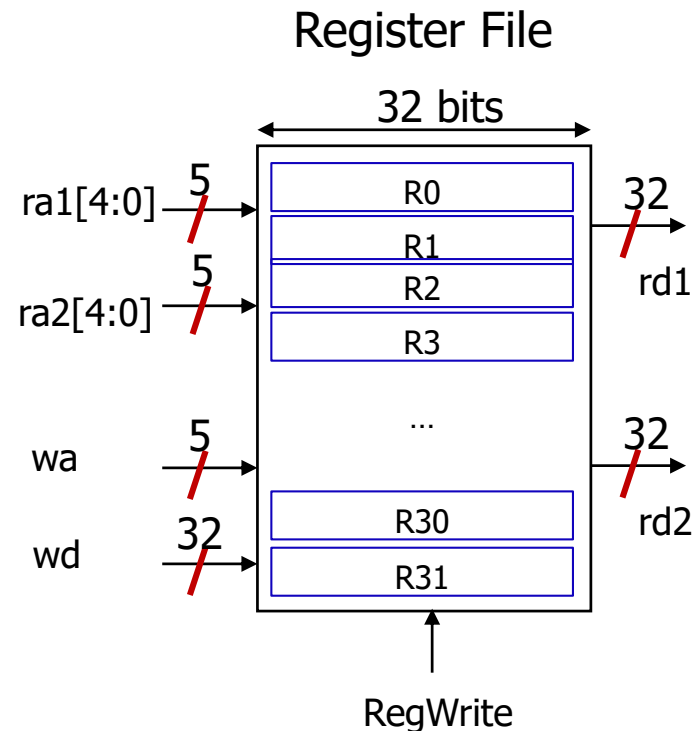
    reg [31:0] rf[31:0];

    // three ported register file
    // read two ports combinatorially
    // write third port on rising edge of clock
    // register 0 hardwired to 0

    always @(posedge clk)
        if (RegWrite) rf[wa] <= wd;

    assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
    assign rd2 = (ra2 != 0) ? rf[ra2] : 0;

endmodule
```



Sign & Zero Extension in Verilog

Why declares it as reg?

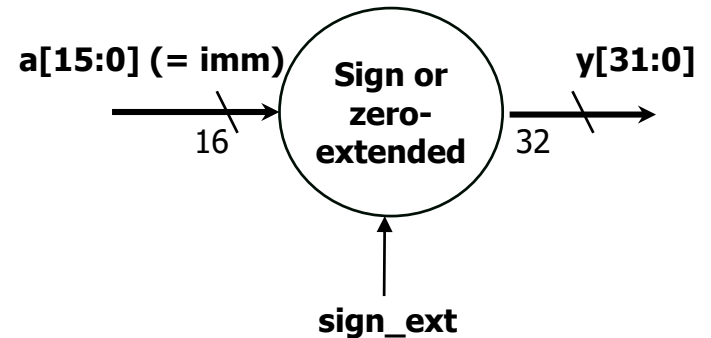
Is it going to be synthesized as registers?

Is this logic combinational or sequential logic?

```
module sign_zero_ext(input      sign_ext,
                    input [15:0] a,
                    output reg [31:0] y);

    always @(*)
    begin
        if (sign_ext) y <= {{16{a[15]}}, a};
        else          y <= {{16{1'b0}}, a};
    end

endmodule
```

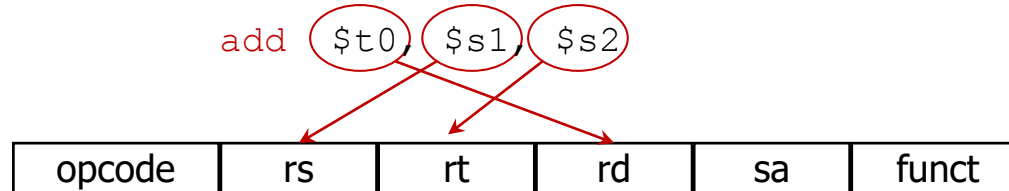


Instruction Execution #1

- Execution of the arithmetic and logical instructions

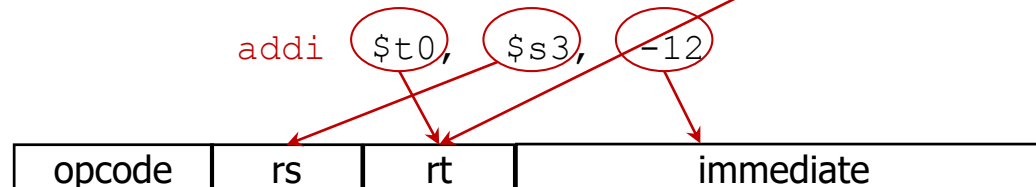
- **R-type** arithmetic and logical instructions

- Examples: `add`, `sub`, `and`, `or` ...
- 2 source operands from the register file



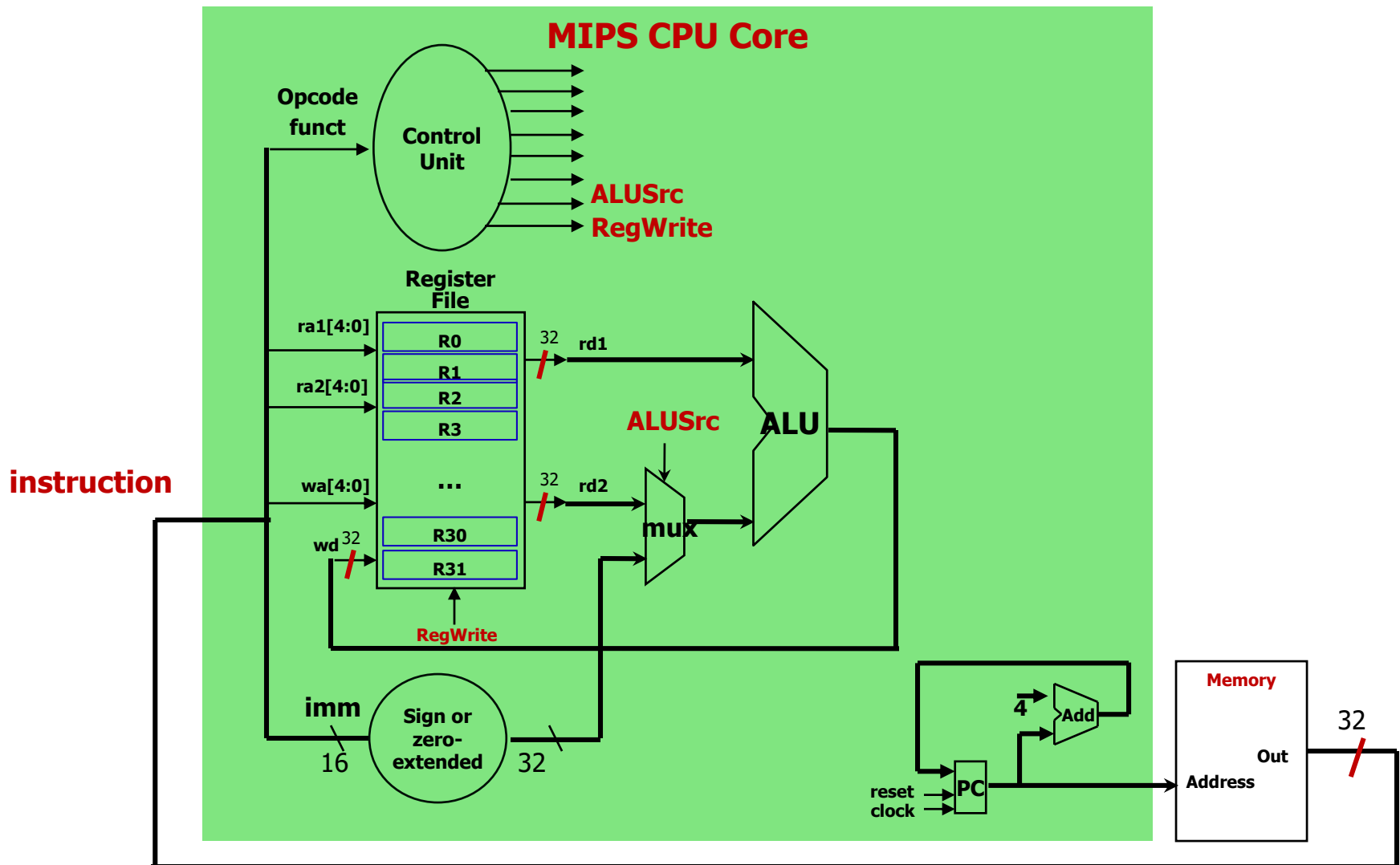
- **I-type** arithmetic and logical instructions

- Examples: `addi`, `andi`, `ori` ...
- 1 source operand from the register file
- 1 source operand from the immediate field



destination register

Schematic with Instruction Execution #1



How to Design Mux in Verilog?

```
module mux2 (input [31:0] d0,
             input [31:0] d1,
             input      s,
             output [31:0] y);

    assign y = s ? d1 : d0;

endmodule
```

OR

```
module mux2 (input [31:0] d0,
             input [31:0] d1,
             input      s,
             output reg [31:0] y);

    always @(*)
    begin
        if (s)    y <= d1;
        else      y <= d0;
    end

endmodule
```

Design it with parameter, so that
this module can be used
(instantiated) in any sized
muxes in your design

```
module mux2 #(parameter WIDTH = 8)
    (input  [WIDTH-1:0] d0, d1,
     input                        s,
     output [WIDTH-1:0] y);
    assign y = s ? d1 : d0;
endmodule
```

```
module datapath(.....);

    wire [31:0] writedata, signimm;
    wire [31:0] srcb;
    wire          alusrc

    // Instantiation
    mux2 #(32) srcbmux(
        .d0 (writedata),
        .d1 (signimm),
        .s  (alusrc),
        .y  (srcb));

endmodule
```

Instruction Execution #2

- Execution of the memory access instructions
 - `lw, sw` instructions

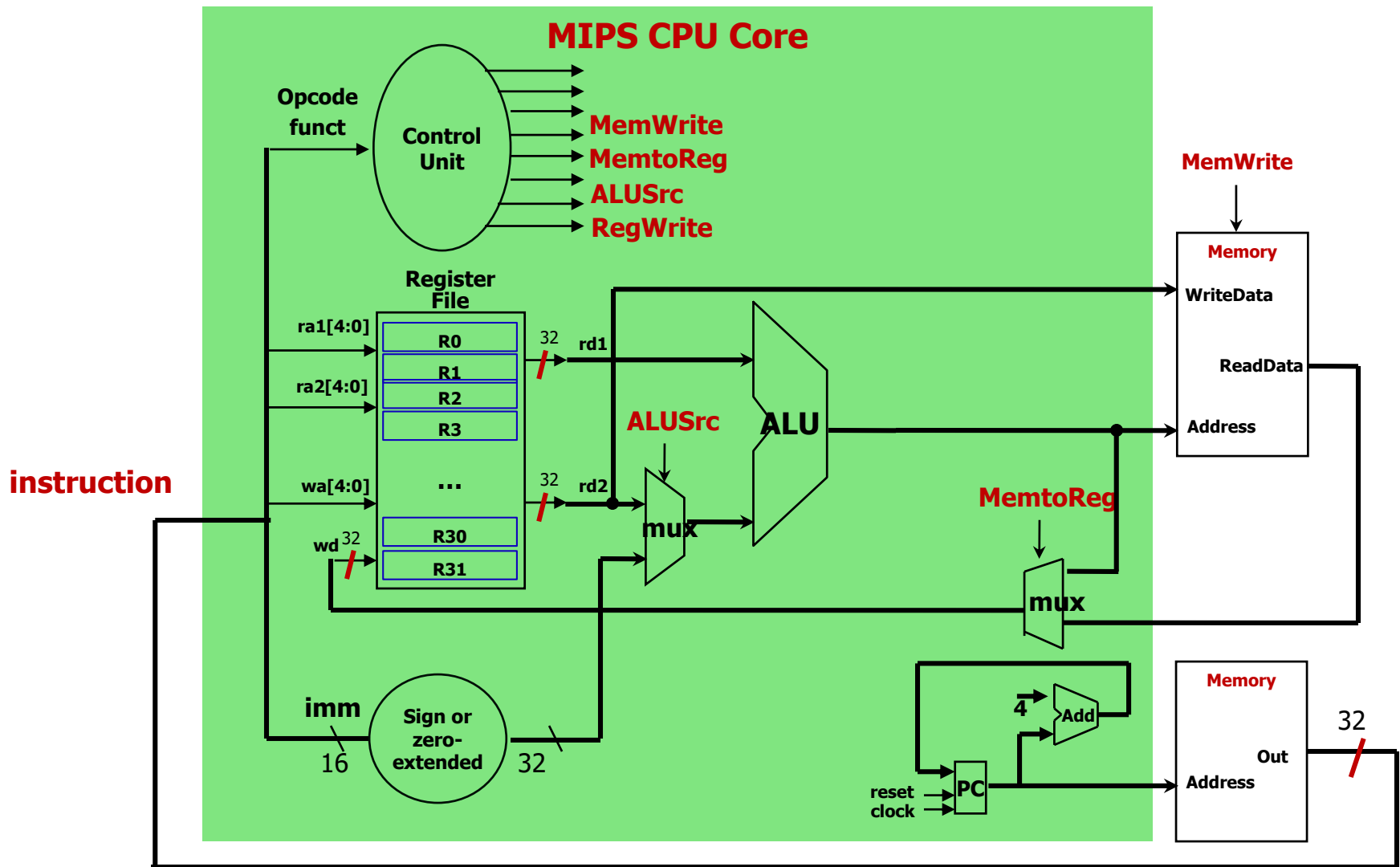
`lw $t0, 24($s3) // $t0 <= [$s3 + 24]`



`sw $t2, 8($s3) // [$s3 + 8] <= $t2`



Schematic with Instruction Execution #2

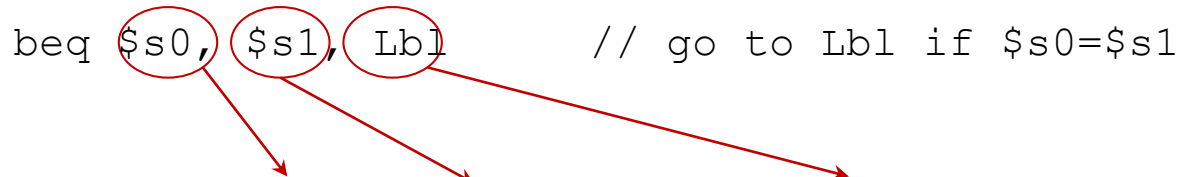


```
lw    $t0, 24($s3) // $t0 <= [$s3 + 24]
sw    $t2, 8($s3)  // [$s3 + 8] <= $t2
```

Instruction Execution #3

- Execution of the branch and jump instructions
 - beq, bne, j, jal, jr instructions

beq \$s0, \$s1, Lbl // go to Lbl if \$s0=\$s1



opcode	rs	rt	immediate
--------	----	----	-----------

Destination = (PC + 4) + (imm << 2)

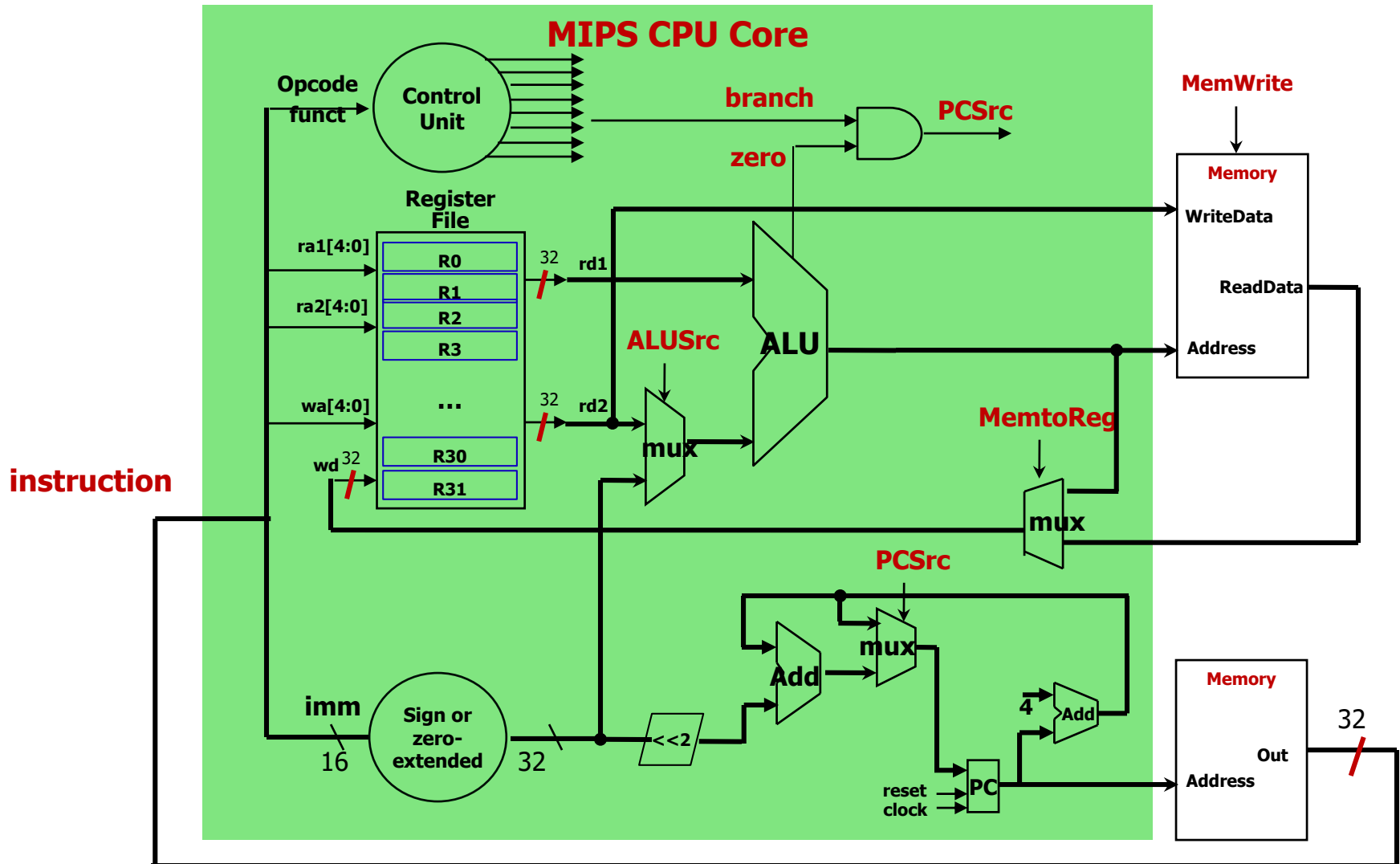
j target // jump



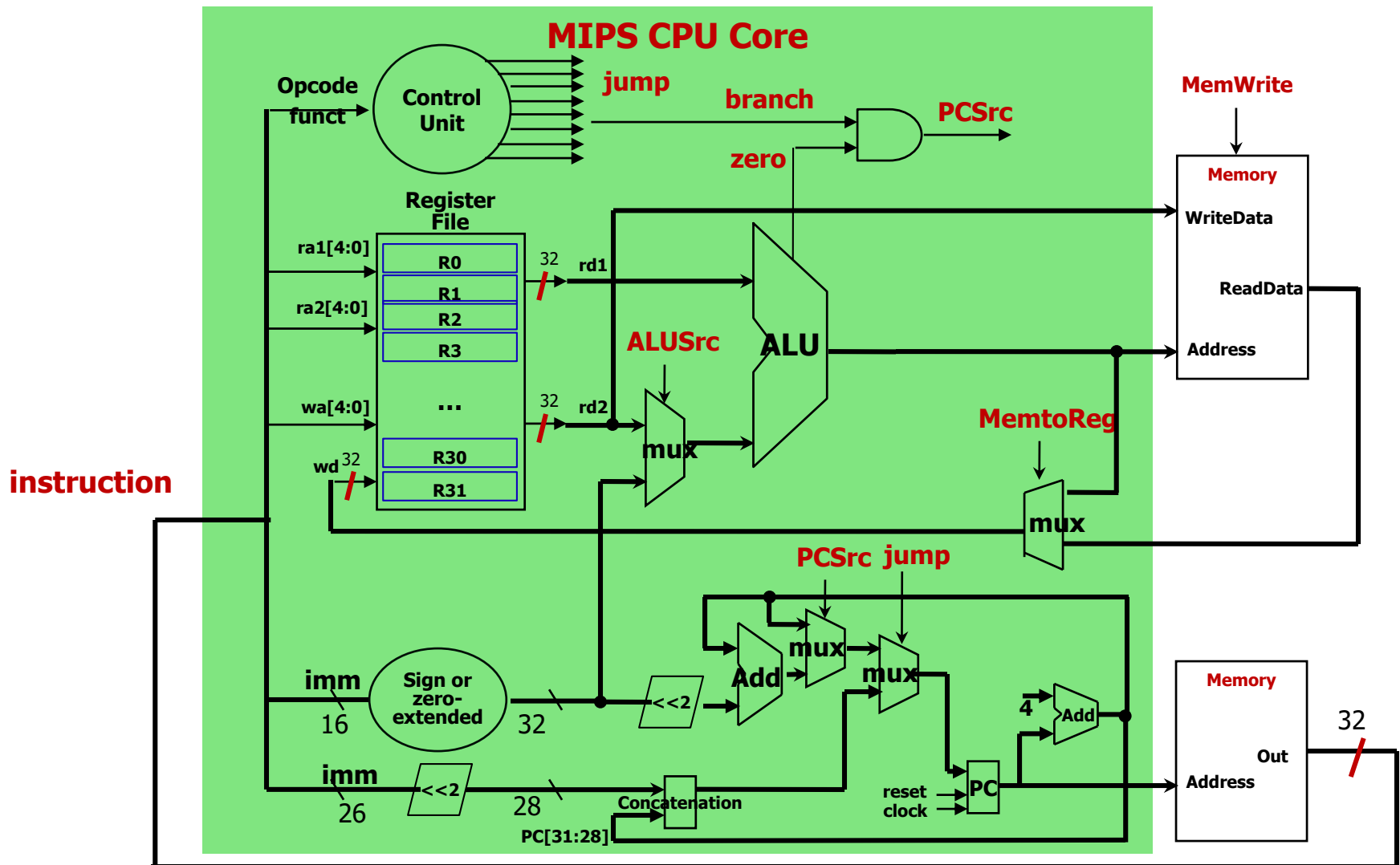
opcode	jump target
--------	-------------

Destination = {(PC+4)[31:28] , jump target, 2'b00}

Schematic with Instruction Execution #3 (beq)



Schematic with Instruction Execution #3 (j)



Demo

- Synthesis with Quartus-II
- Simulation with ModelSim

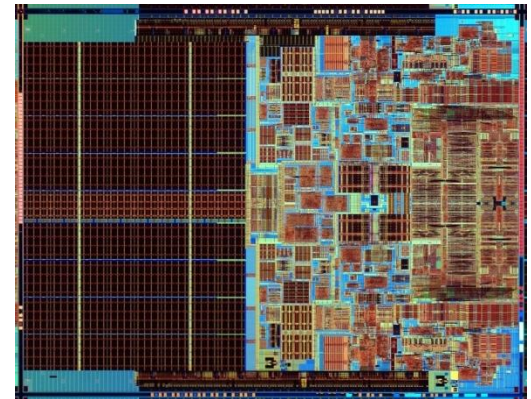
Backup Slides

Why HDL?

- In old days (~ early 1990s), hardware engineers used to draw schematic of the digital logic, based on Boolean equations, FSM, and so on...
- But, it is **not virtually possible** to draw schematic as the hardware complexity increases

- Example:

- Number of transistors in Core 2 Duo is roughly 300 million
- Assuming that the gate count is based on 2-input NAND gate, (which is composed of 4 transistors), do you want to draw 75 million gates by hand? **Absolutely NOT!**



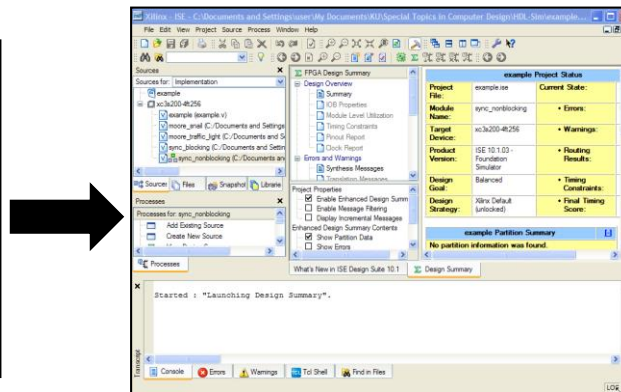
Why HDL?

- Hardware description language (HDL)
 - Allows designer to specify logic function using language
 - So, hardware designer only needs to specify the target functionality (such as Boolean equations and FSM) with language
 - Then a computer-aided design (CAD) tool produces the optimized digital circuit with logic gates
 - Nowadays, most commercial designs are built using HDLs

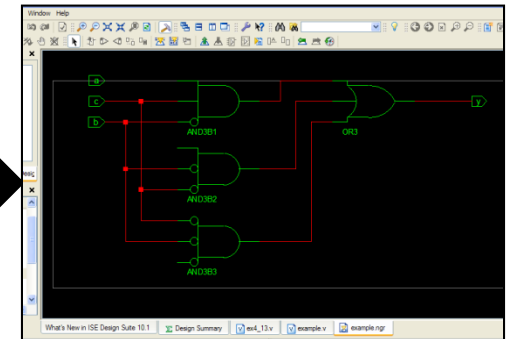
HDL-based Design

```
module example(  
    input  a, b, c,  
    output y);  
  
    assign y = ~a & ~b & ~c |  
              a & ~b & ~c |  
              a & ~b & c;  
  
endmodule
```

CAD Tool



Optimized Gates



HDLs

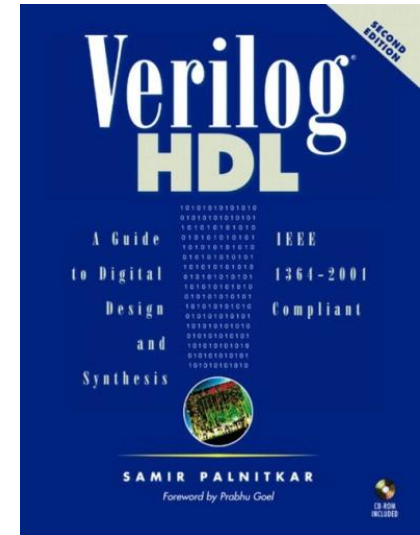
- Two leading HDLs

- **Verilog-HDL**

- Developed in 1984 by Gateway Design Automation
 - Became an IEEE standard (1364) in 1995
 - **We are going to use Verilog-HDL in this class**
 - The book on the right is a good reference (but not required to purchase)

- **VHDL**

- Developed in 1981 by the Department of Defense
 - Became an IEEE standard (1076) in 1987



- IEEE: Institute of Electrical and Electronics Engineers is a professional society responsible for many computing standards including WiFi (802.11), Ethernet (802.3) etc

HDL to (Logic) Gates

- There are 3 steps to design hardware with HDL
 - 1. Hardware design with HDL**
 - Describe your hardware with HDL
 - When describing circuits using an HDL, **it's critical to think of the hardware the code should produce**
 - 2. Simulation**
 - Once you design your hardware with HDL, you need to verify if the design is implemented correctly
 - Input values are applied to your design with HDL
 - Outputs checked for correctness
 - Millions of dollars saved by debugging in simulation instead of hardware
 - 3. Synthesis**
 - Transforms HDL code into a **netlist**, describing the hardware
 - Netlist is a text file describing a list of logic gates and the wires connecting them

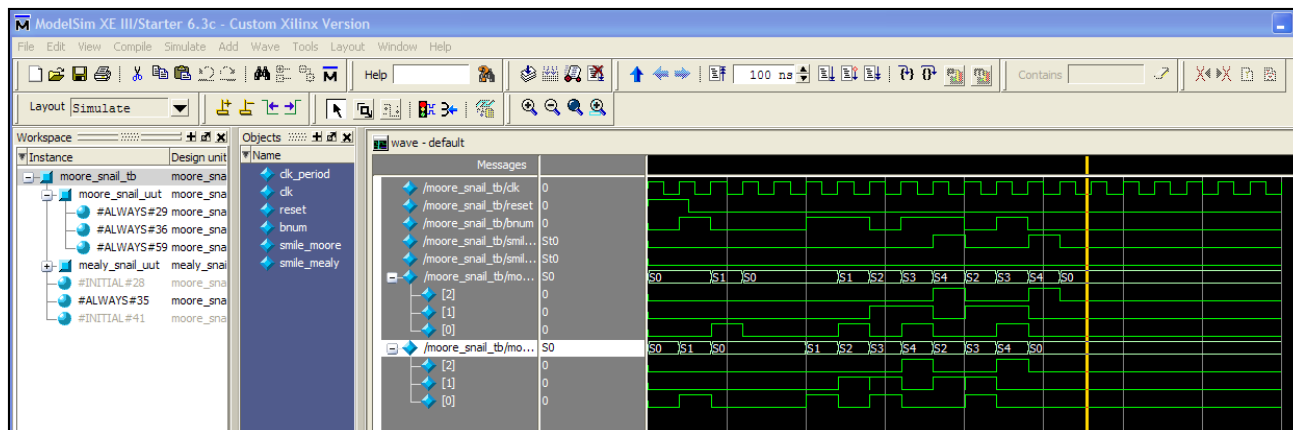
CAD tools for Simulation

- There are renowned CAD companies that provide HDL simulators
 - Cadence
 - www.cadence.com
 - Synopsys
 - www.synopsys.com
 - Mentor Graphics
 - www.mentorgraphics.com
 - We are going to use **ModelSim Altera Starter Edition** for simulation
 - <http://www.altera.com/products/software/quartus-ii/modelsim/qts-modelsim-index.html>

cadence™

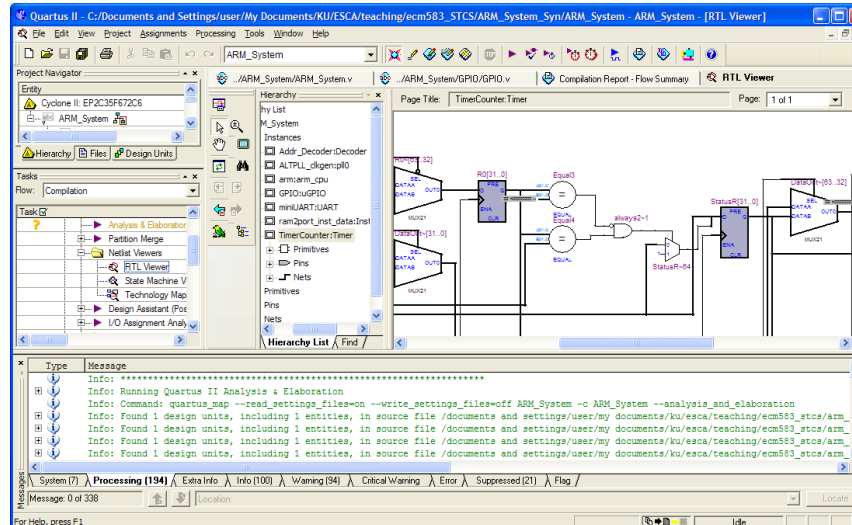
SYNOPSYS®

Mentor
Graphics®



CAD tools for Synthesis

- The same companies (Cadence, Synopsys, and Mentor Graphics) provide synthesis tools, too
 - They are extremely expensive to purchase though
- We are going to use a synthesis tool from Altera
 - **Altera Quartus-II Web Edition (free)**
 - Synthesis, place & route, and download to FPGA
 - <http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html>



MIPS CPU with imem and Testbench

```
module mips_cpu_mem(input clk, reset);

    wire [31:0] pc, instr;

    // instantiate processor and memories
    mips_cpu imips_cpu (clk, reset, pc, instr);
    imem    imips_imem (pc[7:2], instr);

endmodule
```

```
module mips_tb();

    reg      clk;
    reg      reset;

    // instantiate device to be tested
    mips_cpu_mem imips_cpu_mem(clk, reset);

    // initialize test
    initial
    begin
        reset <= 1;
        # 32;
        reset <= 0;
    end

    // generate clock to sequence tests
    initial
    begin
        clk <= 0;
        forever #10 clk <= ~clk;
    end

endmodule
```