5.1 Review: Combinational circuits

Combinational circuits

A ***circuit*** is a connection of electrical components. An ***analog circuit*** has components that transform analog signals, which are signals that take on continuous values, like 0.001, 2.3, or -3.333. A ***digital circuit*** has components that transform digital signals, which are signals that take on only one of a few specific values, like 0 and 1. ("Digit" comes from Latin for finger/toe, of which humans have only a few).

A ***combinational circuit*** is a digital circuit whose output values depend solely on the present combination of input values (past input values do not matter). A combinational circuit can be built from components like AND, OR, and NOT gates, known as ***logic gates***.

PARTICIPATION ACTIVITY 5.1.1: Combinational circuits can be built from AND, OR, and NOT gates.

Animation captions:

1. AND only outputs 1 when all inputs are 1.
2. OR outputs 1 when any input is 1. NOT inverts the input from 0 to 1, or from 1 to 0.
3. A combinational circuit can be built from AND, OR, and NOT gates. Here, w is 1 if xy are 10, or if xy are 11.

PARTICIPATION ACTIVITY 5.1.2: Combinational logic gates.

1) AND with inputs 0 1 outputs ___.
   - 0
   - 1

2) AND with inputs 1 1 outputs ___.
   - 0
   - 1

3) OR with inputs 1 1 outputs ___.
   - 0
   - 1

4) NOT with input 1 outputs ___.
   - 0
   - 1

5) A circuit is AND followed by NOT. If inputs are 0 0, the output is ___.

- 0
- 1

6) A circuit has a b as inputs to an AND gate. Another gate ORs that gate's output with input c. If inputs a b c are 0 0 1, the output of the OR is ___.
   - 0
   - 1

Combinational behavior: Truth tables and equations

A ***truth table*** is one way to describe the behavior of a combinational circuit, listing the output value for every possible combination of input values. A ***Boolean equation*** is another way to describe the behavior of a combinational circuit, using operators AND (a*b or just ab), OR (a+b), and NOT (a'). Boolean refers to algebra involving only two values: true or false. Designers commonly describe desired circuit behavior using a truth table or equation, and then derive a circuit.

PARTICIPATION ACTIVITY 5.1.3: Truth tables, Boolean equations, and combinational circuits.

Animation captions:

1. A truth table lists all possible combinations of input values, and the output value for each combination.
2. An equation describes the same behavior, in a more compact form.
3. An equation is easily converted to a circuit.

PARTICIPATION ACTIVITY 5.1.4: Truth tables, Boolean equations, and circuits.

1) 3 inputs yield a truth table with ___ possible input value combinations.
   - 6
   - 8

2) 8 inputs yield a truth table with ___ possible input values.
   - 16
   - 256

3) A truth table with inputs x, y, z has output f = 1 for row 0 0 1. The equation will thus include term _____.
   - x'y'z
   - xyz'

4) A truth table with inputs x, y, z has 8 rows. The first 5 rows set output f = 1. If an equation is written directly from the

table, the equation will start with f = x'y'z' + ... How many terms will the equation have? (x'y'z' is one term).

- 3
- 5

5) A truth table with inputs x, y has f = 1 for row 0 1 and for row 1 1. The equation is thus f = x'y ___ xy.

- +
- *

6) An equation is f = xy'z' + xyz. How many AND and OR gates are in the circuit?

- 2
- 3

In an equation, a ***minterm*** is a term that involves all a function's variables exactly once. An equation derived by ORing terms for each 1 in a truth table is called a ***sum-of-minterms***.

CHALLENGE ACTIVITY 5.1.1: Convert the table to a sum-of-minterms.

459784.3174716.qx3zqy7

CHALLENGE ACTIVITY 5.1.2: Convert the equation to a circuit.

Instructions: Click a logic gate to add the logic gate into the circuit. To connect two terminals, click and drag the mouse pointer from one terminal to the other terminal.

459784.3174716.qx3zqy7

5.2 Review: Decoders, muxes, and adders

Multiplexors (mux)

A ***multiplexor*** is a combinational circuit that passes one of multiple data inputs through to a single output, selecting which one based on additional control inputs. ***Mux*** is short for multiplexor. A mux's control inputs are called ***select lines***.

Analogy: Due to road construction, four lanes (the data inputs) may be reduced to a single lane (the single output). A policeman (the select inputs) selects which one lane currently passes through by blocking the other lanes.

A ***4x1 mux***, spoken as "4 to 1 mux", has 4 data inputs, 1 data output, and requires 2 select inputs.

PARTICIPATION ACTIVITY 5.2.1: 4x1 mux.

Animation captions:

1. s1 = 0, s0 = 0 (s1s0 = 00) allows i0 to pass through to y.
2. s1s0 = 01 allows i1 to pass.
3. Whatever i1's value is will pass through.
4. s1s0 = 10 allows i2 to pass.
5. s1s0 = 11 allows i3 to pass.

PARTICIPATION ACTIVITY 5.2.2: Muxes.

Assume i3 = 0, i2 = 1, i1 = 1, and i0 = 0.

1) If s1 = 0 and s0 = 0, then y = ?
   - 1
   - 0

2) If s1 = 1 and s0 = 0, then y = ?
   - 1
   - 0

3) s1s0 = ? allows i0 to pass through to y.
   - 10
   - 00

4) s1s0 = ? allows i3 to pass through to y.
   - 11
   - 10

Decoders

A ***decoder*** is a combinational circuit that converts N inputs to a 1 on one of 2^N outputs. A ***2x4 decoder***, spoken as "2 to 4 decoder", converts two inputs to a 1 on one of four outputs.

As an analogy, consider a four-room high-school with a microphone in the principal's office and speakers in each room. By configuring two switches, the principal can specify which one room will hear her voice.

PARTICIPATION ACTIVITY 5.2.3: A 2x4 decoder.

Animation captions:

1. i1i0 = 00: y0 = 1 (all other 0s).
2. i1i0 = 01: y1 = 1 (all other 0s).
3. i1i0 = 10: y2 = 1 (all other 0s).
4. i1i0 = 11: y3 = 1 (all other 0s).

PARTICIPATION ACTIVITY 5.2.4: 2x4 decoder.

Given a 2x4 decoder.

1) If i1i0 = 00, then y1 = ?

Check **Show answer**

2) If i1i0 = 01, then y1= ?

Check **Show answer**

3) i1i0 = ?? configures the decoder to output y0 = 0, y1 = 0, y2 = 0, and y3 = 1.

Check **Show answer**

4) How many outputs are set to 1 at any given time?
Type: 0, 1, 2, 3, or 4

Check **Show answer**

5) i1i0 = ?? configures the decoder to output y0 = 0, y1 = 0, y2 = 1, and y3 = 1.
Type: 00, 01, 10, 11, or ** if not possible.

Check **Show answer**

Adders

An ***adder*** is a combinational circuit that adds two N-bit inputs, producing an N-bit sum output. Ex: Given 001 and 011, a 3-bit adder produces output 100 (in other words, 1 + 3 produces 4, assuming unsigned numbers). Some adders may include a carry-out output bit, so 110 + 100 produces a carry-out of 1 and a sum of 010.

PARTICIPATION ACTIVITY 5.2.5: A 3-bit adder.

Animation captions:

1. An N-bit adder adds two N-bit numbers, producing an N-bit sum. Here, a 3-bit adder adds 001 + 011, producing 100 (so 1 + 3 produces 4).
2. The multi-bit inputs and outputs are typically named with a single letter like A, and multiple wires are drawn as a single thicker wire.
3. With some additional internal logic, the adder can also perform subtraction when a sub control input is 1.

PARTICIPATION ACTIVITY 5.2.6: Adder.

1) In the above 3-bit adder, if A = 010 and B = 100, what is S?

Check **Show answer**

2) In the above 3-bit adder, if A = 111 and B = 001, what is S?

Check **Show answer**

3) In the above 3-bit adder/subtractor, what should sub be set with to perform addition?

Check **Show answer**

CHALLENGE ACTIVITY 5.2.1: Muxes, decoders and adders.

459784.3174716.qx3zqy7

5.3 Review: Timing diagrams

A ***timing diagram*** graphically shows a circuit's output values for given input values that change over time. Each signal (input or output) name is listed on the left. Time proceeds to the right. Each signal is drawn as a high line (1) or a low line (0).

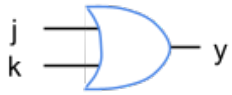
PARTICIPATION ACTIVITY 5.3.1: Timing diagram for an AND gate.

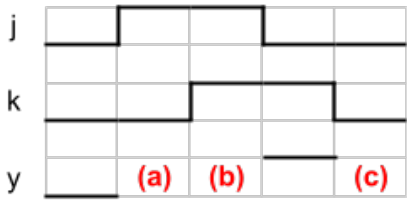
Animation captions:

1. If a = 0 and b = 0, then y = 0.
2. If a = 1 and b = 0, then y = 0.
3. If a = 0 and b = 1, then y = 0.
4. If a = 1 and b = 1, then y = 1.

PARTICIPATION ACTIVITY 5.3.2: Logic gates: NOT, AND, and OR.

Complete the timing diagram.





1) (a)

- 0
- 1

2) (b)

- 0
- 1

3) (c)

- 0
- 1

CHALLENGE ACTIVITY 5.3.1: Indicate y's value over time.

Click on the output waveform to adjust that waveform.

459784.3174716.qx3zqy7

5.4 Review: Registers

A ***sequential circuit*** is a digital circuit whose output values may depend on present *and past* input values (vs. depending only on present values in a combinational circuit). To depend on past values, a sequential circuit must

somehow store one or more bits.

An N-bit ***register*** (such as an 8-bit register) stores N bit values, loading new bit values when a clock input rises if a load input is 1. A load input (ld) indicates when the register should be loaded. A reset input (rst) indicates that the register's bits should be reset to 0 (having priority over ld).

All a circuit's registers may share one ***clock*** signal, whose ***rising edge*** (the instant a 0 changes to 1) synchronizes loading of all registers (like a drum beat synchronizes a marching band).

PARTICIPATION ACTIVITY 5.4.1: Registers.

Animation captions:

1. The reset input is set with 1 when the system is powered on, resetting the stored bits to 000, which then appear at the outputs.
2. When the clock rises, if ld is 0, the values on d2d1d0 are ignored. Here, the register continues to store 000.
3. When the clock rises, if ld is 1, the data input values are stored in the register, and appear at the outputs.
4. Those bits remain stored, even though the data inputs may change (until a future rising clock with ld = 1).
5. Multiple wires are typically drawn on the block diagram using one thicker wire, and on the timing diagram as values within a rectangle.
6. A single clock signal typically synchronizes the loads of all registers in a design.

PARTICIPATION ACTIVITY 5.4.2: Registers.

Consider the above 3-bit load register.

1) What are q2q1q0 when rst = 1 and ld = 1, assuming q2q1q0 were previously 110, and d2d1d0 are 111?
   - ○ 110
   - ○ 000
   - ○ 111
   - ○ Depends on clk's value

2) Assume rst = 0, ld = 1, d2d1d0 are 110, and q2q1q0 are 111. When a rising clock occurs, what do q2q1q0 become?
   - ○ 111
   - ○ 110
   - ○ 000

3) Assume rst = 0, ld = 0, d2d1d0 are 110, and q2q1q0 are 111. When a rising clock occurs, what do q2q1q0 become?
   - ○ 111

- 110
- 000

4) Assume Y connects to register R1's inputs, and R1's outputs connect to R2's data inputs. On rising clock 1, Y was 100. On rising clock 2, Y was 000. On rising clock 3, Y was 111. What is now in R2?
   - 100
   - 000
   - 111

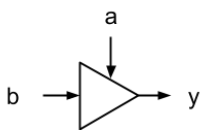
CHALLENGE ACTIVITY 5.4.1: Indicate Z's value over time.

459784.3174716.qx3zqy7

5.5 Review: Register files

A ***three-state buffer*** is a component that outputs y = b if input a is 1, and outputs Z if a is 0. Z represents an electrical situation known as ***high impedance***. A wire with Z is akin to no wire. As such, three-state buffers support efficient combining of multiple wires into one wire, as will be seen when introducing register files below. Of course, only one connected wire can have a non-Z (0 or 1) value, else the values collide. Further details on three-state buffer and high impedance are beyond this material's scope.

Figure 5.5.1: Three-state buffer.



a	y
0	'Z' (high impedance)
1	b

PARTICIPATION ACTIVITY 5.5.1: Three-state buffer.

Given a three-state buffer with control input a, data input b, and data output y.

1) If a = 1 and b = 1, then y = ?
   - 1
   - 'Z'

2) If a = 1 and b = 0, then y = ?

- 1
3) If a = 0 and b = 0, then y = ?
- 0
- 0
- 'Z'

A three-state buffer is more commonly called a ***tri-state buffer***, but that name is trademarked. The term *three-state driver* (or *tri-state driver*) is also used.

An NxM ***register file*** efficiently implements access to N M-bit registers. A 16x32 register file has 16 registers, each 32 bits wide. To avoid having 512 (16 times 32) wires connecting with those registers for loads, a register file consolidates loads through one 32-bit data input. Of course, the tradeoff is that only one register can be loaded at a time, as indicated by a 4-bit address input. Loading a register in a register file is known as a ***write*** operation. The data input, address input, and load enable input for writing are together called a ***write port***. Similarly, a register file consolidates read wires into a single 32-bit data output, 4-bit address input, and enable input, forming a ***read port***. The read port may use three-state buffers to efficiently connect the data wires.

PARTICIPATION ACTIVITY 5.5.2: 4x8 register file.

Animation captions:

1. A 4x8 register file has 4 registers, each 8-bits wide.
2. The write port consists of a write address, a write enable control signal, and write data.
3. W_addr determines which register is enabled for load on the rising clock edge. W_en must be set to 1 to load the register.
4. If W_en is set to 0, no register is loaded on the rising clock edge.
5. The read port consists of a read address, a read enable control signal, and read data. RA_addr determines which register is read. RA_en must be set to 1 to read the register.
6. A register file commonly has multiple read ports.
7. Writes are synchronous, but reads asynchronous. Reads starts right away. But a write occurs on a rising clock. Shortly after, the newly-written value appears at the read's output port.
8. A register file is commonly represented as a block symbol.

PARTICIPATION ACTIVITY 5.5.3: Register file.

1) A 32x8 register file consists of ____ registers.
- 8
- 32
- 40

2) The write address, write enable control, and write data are collectively known as what?

3) W_en must be set to 1 to write to, or load a register.
   - Write port
   - Read port
   - True
   - Write decoder
   - False

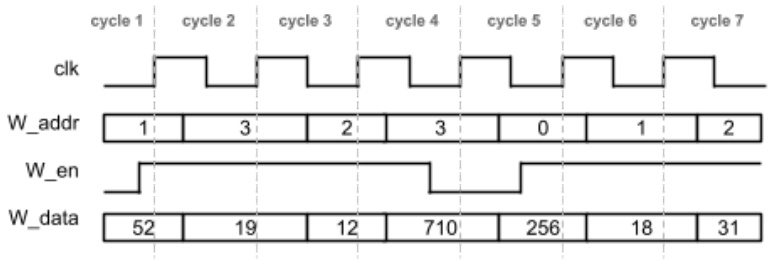
4) Given a 32x8 register file, how many bits is W_addr?
   - 2
   - 3
   - 5

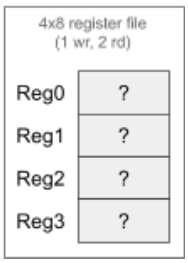
5) Assuming a register file with one write port and two read ports, write and read operations can occur simultaneously.
   - True
   - False

6) Assuming a register file with one write port and two read ports, two read operations can occur simultaneously.
   - True
   - False

PARTICIPATION ACTIVITY 5.5.4: Register files: Writing.

For the given values of W_addr, W_en, and W_data, indicate the register file's contents in a given clock cycle.





1) Reg1, cycle 2

Check **Show answer**

2) Reg1, cycle 6

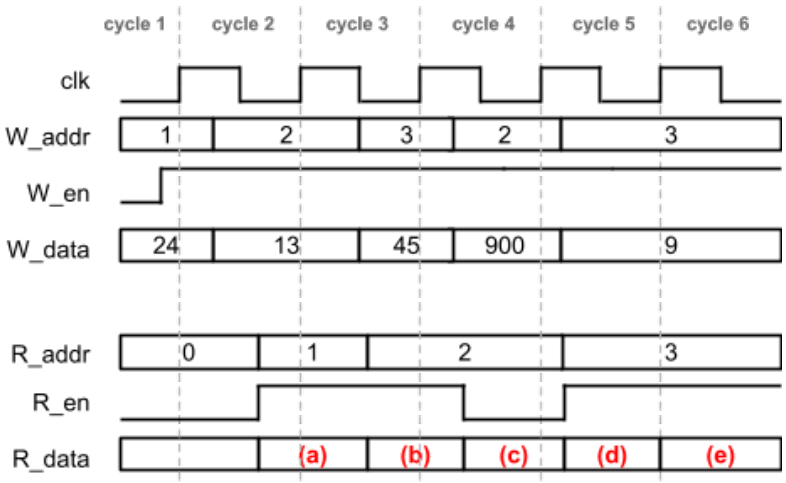
Check **Show answer**

3) Reg3, cycle 5

Check Show answer

PARTICIPATION ACTIVITY 5.5.5: Register files: Writing and reading.

Assume a 4x32 register file with one read port and one write port. Determine R_data's value at each indicated time. If appropriate, type: Z.



1) (a)

Check Show answer

2) (b)

Check Show answer

3) (c)

Check Show answer

4) (d)

Check Show answer

5) (e)

Check Show answer

CHALLENGE ACTIVITY 5.5.1: Register file writing and reading.

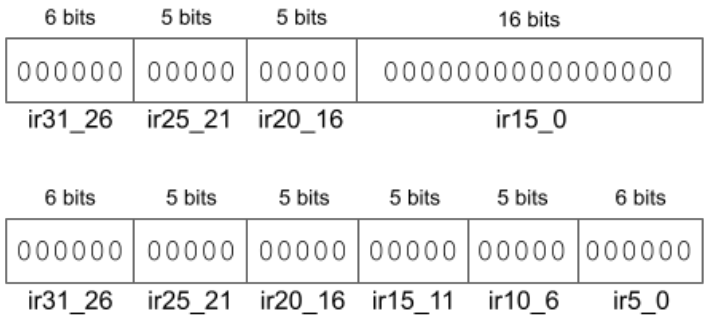
459784.3174716.qx3zqy7

5.6 Base MIPSzy (lw, sw, addi, add): Behavior

MIPSzy instruction field bits

When designing a digital circuit to implement the MIPSzy processor, for ease of reference, bits forming an instruction field are given a name, like ir31_26 referring to bits 31, 30, 29, 28, 27, and 26.

Figure 5.6.1: Convenient names for bit groups in a machine instruction, corresponding to fields in different instruction types.



PARTICIPATION ACTIVITY 5.6.1: Instruction field bits.

Consider the figure above.

1) Leftmost 6 instruction bits.
   - ir31_26
   - ir31_25
   - ir5_0

2) Rightmost 6 instruction bits.
   - ir5_0
   - ir15_0

3) The rightmost 16 bits are named ir15_0, but those do not include the rightmost 6 bits, which already have a name.

- ○ True
- ○ False

Base MIPSzy behavior

For simplicity, this section defines the behavior of a base MIPSzy version. The ***base MIPSzy*** version only implements the lw, sw, addi, and add instructions. Other MIPSzy instructions will later be added to the base version.

A ***processor*** is a circuit that executes instructions. Designers commonly first described a processor's desired behavior, and then derive a circuit to implement that behavior. Such behavior can be described using a C-like language, as below (that C-like description of the processor should not be confused with the program that the processor will execute).

The behavioral description declares key storage components as variables: instruction memory (IM), data memory (DM), register file (RF), and Program Counter (PC). The ***Program Counter (PC)*** holds the address of the current instruction to execute.

PARTICIPATION ACTIVITY 5.6.2: Base MIPSzy's behavioral description.

Animation captions:

1. The behavioral description for MIPSzy repeats an Execute state every clock cycle.
2. Each clock cycle, the Execute state gets the current instruction and carries out that instruction's actions (see figure below for Execute's full actions).

Figure 5.6.2: MIPSzy behavioral description: The Execute state's actions described using C-like code.

```
ir = IM[PC/4];
PC = PC + 4;

// Assume ir31_26 etc are extracted

if      (ir31_26 == 0b100011)   { // lw
  dm_a = RF[ir25_21];
  dm_rd = DM[(dm_a-4096)/4];
  RF[ir20_16] = dm_rd;
}
else if (ir31_26 == 0b101011)   { // sw
  dm_a = RF[ir25_21];
  dm_wd = RF[ir20_16];
  DM[(dm_a-4096)/4] = dm_wd;
}
else if (ir31_26 == 0b001000)   { // addi
  add1 = RF[ir25_21];
  add2 = ir15_0;
  RF[ir20_16] = add1 + add2;
}
else if ( (ir31_26 == 0b000000)
       && (ir5_0 == 0b100000) ) { // add
  add1 = RF[ir25_21];
  add2 = RF[ir20_16];
  RF[ir15_11] = add1 + add2;
}
```

- ir = IM[PC/4] fetches the current instruction. PC = PC + 4 readies the PC for the next fetch.
- The if-else determines if the instruction opcode is lw, sw, addi, or add.
- Each if-else branch carries out that instruction's actions
- Variables like dm_a or add1 used as temporary values within the Execute state will become wires.
- IM and DM are 1024-word memories, 4 bytes per word, yielding 4096 items each. MIPSzy only supports word access though, so the rightmost two address bits are ignored (hence the "/4" when accessing IM or DM).
- DM is implemented as a separate memory from IM. Hence, DM addresses are first adjusted by subtracting 4096. Ex: Address 4096 becomes address 0 for DM, 5000 becomes 4, etc.. (And then /4 is performed as described above).

Table 5.6.1: Reminder: Base MIPSzy machine instructions.

Assembly	Machine
lw $t0, 0($t1)	100011 01001 01000 0000000000000000
sw $t0, 0($t1)	101011 01001 01000 0000000000000000
addi $t0, $t1, 15	001000 01001 01000 0000000000001111
add $t0, $t1, $t2	000000 01001 01010 01000 00000 100000

PARTICIPATION ACTIVITY 5.6.3: MIPSzy behavior description: The Execute state's actions.

Consider the MIPSzy behavioral description figure above, showing the MIPSzy Execute state's actions using C-like code.

1) The Execute state first reads the current instruction from IM, into a variable named ir.
   - True
   - False

2) The Execute state increments PC by 4 before reading the IM.
   - True
   - False

3) The figure's C-like code shows how field names like ir31_26 are associated with ir.
   - True
   - False

4) The if-else statement compares ir31_26 with the opcodes for lw, sw, addi, and add.
   - True
   - False

5) If ir31_26 is 001000, the Execute state

will sum the values from two registers.

- True
- False

6) addi's actions include: `add1 = RF[ir25_21];` That statement reads RF using bits ir25_21 as the register address.

- True
- False

7) addi's actions use variables add1 and add2. Those variables will become registers in the processor circuit.

- True
- False

8) The add instruction's actions write to RF[ir15_11].

- True
- False

9) Because DM is implemented in a separate memory as IM, addresses intended for DM have 4096 subtracted first.

- True
- False

Note to instructors: Word-alignment.

For the authors, defining MIPSzy involved many tradeoffs between goals like simplicity (to assist learning) and MIPS consistency (for simulator use and smooth lead-in to a next course). Keeping addressing consistent was critical for the latter goal, but unfortunately requires the various address adjustments that appear above (like having a 1024-word memory rather than 4096, and like dividing by 4 to ignore byte addresses).

5.7 Base MIPSzy: Processor design

The base MIPSzy's processor design will include several components: instruction memory (IM), data memory (DM), register file (RF), adder (ADD), and control logic (CTRL). To implement the base MIPSzy processor, a designer can create a circuit of components that supports the MIPSzy behavioral description's actions, and then convert the behavioral description into control logic actions, implemented as a combinational circuit in CTRL, that configure the other components to carry out the current instruction's actions.

PARTICIPATION ACTIVITY 5.7.1: Creating a processor circuit that supports the MIPSzy behavioral description's actions.

Animation captions:

1. ir = IM[PC/4] is supported by connecting PC's output to IM's address input a. IM's data output d is named ir. Only bits 11:2 of PC's 32-bit output are connected to IM, due to MIPSzy being simpler than MIPS.
2. PC = PC + 4 is supported using the "+ 4" component that increments by 4.
3. Extracting ir sub-bits is just done with wires.
4. The bitwise equality comparison of ir31_26 with the opcode will be done in the control logic.
5. Reading RF[ir25_21] is supported by connecting ir25_21 to RF's read port 1's address input, r1a. RF read port 1's data output is labeled dm_a.
6. dm_a is connected to DM's address input a. DM's read data output is labeled dm_rd.
7. Each action is examined similarly, and connections added as needed. If two or more items connect to one input, a mux is added.

In addition to key components (IM, RF, ADD, DM, PC), the processor's circuit also includes muxes wherever a component's input comes from two different sources (to support different instructions); the control logic will set the mux's select line to pass the correct source for the current instruction. Also, the processor's circuit includes a ***sign-extender*** (SE) component, which extends a two's-complement binary number into a wider number by prepending 1's (if the leftmost bit was 1) or 0's (if the leftmost bit was 0), in this case extending from the instruction's 16-bit immediate to a 32-bit input for the adder.

PARTICIPATION ACTIVITY 5.7.2: Creating a circuit to support the MIPSzy behavioral description's actions.

1) The PC holds the current instruction address. To support fetching the current instruction, the PC's output is connected to _____.
   - ○ DM's address input
   - ○ IM's address input
   - ○ RF's address input

2) The PC has _____ different sources of data input.
   - ○ 1
   - ○ 2
   - ○ 3

3) The PC register's load input is ____.
   - ○ tied to 1
   - ○ set to 1 or 0 by CTRL

4) The comparison of ir31_26 with 100011 is done in the _____ component.

- ADD
- CTRL

5) The lw instruction reads an RF register using RF port _____.
   - 1
   - 2

6) The lw instruction reads DM onto dm_rd (DM read data). That data is written into the RF register specified by what address?
   - ir25_21
   - ir20_16
   - ir15_11

7) The sw instruction reads an address from an RF register similar to lw. However, sw then reads a second RF register on RF read port ____, whose value connects to DM's data input.
   - 1
   - 2

8) The addi instruction connects ir15_0 to ADD's _____ input.
   - top
   - bottom

9) MIPS supports byte addressing, but MIPSzy only supports word addresses, ignoring byte addresses. Thus, MIPSzy's design ignores the rightmost ___ bits of an address.
   - 2
   - 4

10) For simplicity, the MIPSzy design ignores the leftmost bits of memory addresses, considering only bits 11:2. As such, the design _____ IM addresses outside of 0-4092 and DM addresses outside of 4096-8188.
    - detects
    - ignores

11) MIPSzy's high-level behavior subtracted 4096 from data memory addresses due to DM being

implemented in a separate memory from IM. How is such subtraction carried out in MIPSzy's design?

- Using a subtractor
- Using the ADD component
- No such subtraction is performed

Base MIPSzy's control logic actions

The behavioral description's actions can be replaced by control logic actions that carry out the desired high-level actions on the processor circuit, as shown below. Any control signal not explicitly set (with 0 or 1) on a given pass through the Execute state is implicitly set with 0.

Figure 5.7.1: Base MIPSzy behavioral description's Execute state actions, and corresponding control logic actions making use of the above processor circuit.

Behavioral description's actions:

```
ir = IM[PC/4];
PC = PC + 4;
// Assume ir31_26 etc are
extracted
                    // lw
if      (ir31_26 == 0b100011)  {
  dm_a = RF[ir25_21];
  dm_rd = DM[(dm_a-4096)/4];
  RF[ir20_16] = dm_rd;
}
                    // sw
else if (ir31_26 == 0b101011)  {
  dm_a = RF[ir25_21];
  dm_wd = RF[ir20_16];
  DM[(dm_a-4096)/4] = dm_wd;
}
                    // addi
else if (ir31_26 == 0b001000)  {
  add1 = RF[ir25_21];
  add2 = ir15_0;
  RF[ir20_16] = add1 + add2;
}
                    // add
else if ( (ir31_26 == 0b000000)
       && (ir5_0 == 0b100000) ){
  add1 = RF[ir25_21];
  add2 = RF[ir20_16];
  RF[ir15_11] = add1 + add2;
}
```

Control logic actions:

```
// PC11:2 connected to IM's address sets
ir
// PC's ld input tied to 1 loads PC + 4

                    // lw
if      (ir31_26 == 0b100011)  {
   rf_r1e = 1; // Reads RF[ir25_21] onto
dm_a
   dm_re = 1;  // Reads DM using dm_a 11:2
   rf_wd_s = 0;  rf_wa_s = 1;  rf_we  = 1;
}
                    // sw
else if (ir31_26 == 0b101011)  {
   rf_r1e = 1;
   rf_r2e = 1;
   dm_we = 1;
}
                    // addi
else if (ir31_26 == 0b001000)  {
   rf_r1e = 1;
   add_add2_s = 0;
   rf_wd_s = 1;  rf_wa_s = 1;  rf_we = 1;
}
                     // add
else if ( (ir31_26 == 0b000000)
        && (ir5_0   == 0b100000) ){
   rf_r1e = 1;
   rf_r2e = 1;  add_add2_s = 1;
   rf_wd_s = 1;  rf_wa_s = 0;  rf_we = 1;
}
```


The control logic CTRL can be implemented using a standard combinational circuit design process, based on the control logic actions created above.

PARTICIPATION ACTIVITY 5.7.3: Base MIPSzy's control logic actions.

Consider the figure above showing MIPSzy's control logic actions.

1) Which causes ir = IM[PC/4]?
   - ir_ld = 1;
   - IM_re = 1;
   - None

2) Which control logic action causes PC = PC + 4?
   - none
   - plus4 = 1;

3) lw: Which lw control logic action(s) carries out statement RF[ir20_16] = dm_rd?
   - rf_r1e = 1;
   - dm_re = 1;
   - rf_wd_s = 0;
     rf_wa_s = 1;
     rf_we = 1;

4) sw: Which sw control logic action carries out statement DM[(dm_a-4096)/4] = dm_wd?
   - rf_r1e = 1;
   - rf_r2e = 1;
   - dm_we = 1;

5) addi: Which addi control logic action(s) carries out statement add1 = RF[ir25_21]?
   - rf_r1e = 1;
   - add_add2_s = 0;
   - rf_wd_s = 1;
     rf_wa_s = 0;
     rf_we = 1;

6) add: For the add instruction, high-level behavior action add2 = RF[ir20_16] becomes the control actions:
   rf_r2e = 1; add_add2_s = 1;
   Why is add_add2_s set to 1?
   - To pass r2d to the adder
   - To pass ir15_0 to the adder

- To enable the adder

7) Every written control signal in the above MIPSzy control logic actions must be an output of the CTRL component.
   - True
   - False

Executing instructions on the base MIPSzy processor design

The base MIPSzy processor consists of the key components of CTRL, PC, IM, RF, ADD, and DM, as below.

A program would be pre-loaded into IM. Upon starting the processor, power-on reset circuitry (not shown) would set the rst (reset) inputs of PC and RF to 1 for a short duration, clearing those components' storage elements to 0's. PC's 0 reads the instruction at IM[0] to ir, whose opcode flows to the control logic CTRL, which sets appropriate control outputs to carry out the instruction (namely, having the correct values waiting at RF's wd, wa, and we inputs, which take effect on the next rising clock).

PARTICIPATION ACTIVITY 5.7.4: Executing an instruction on the MIPSzy processor: At the end of the clock cycle for addi, the result 5000 is ready to be written into RF's $t6 (01110) on the next cycle.

Animation captions:

1. The machine instructions are preloaded in IM. This program doubles the value in DM location 5000.
2. Upon powering on the system, power on reset circuitry (not shown) sets the rst (reset) inputs of PC and RF to 1 for a short duration, clearing those components' stored bits to 0's.
3. Because PC is now 0, the instruction at IM[0] is read (and the +4 causes 4 to wait at PC's input too).
4. The opcode 001000 flows to CTRL, which detects an addi instruction. CTRL sets the relevant control outputs for addi. All other CTRL outputs are 0's.
5. The encoding for $zero, 00000, flows to r1a, causing $zero's 0 (32 bits) to be read to add1. Simultaneously, ir15_0 (with 0's prepended by SE to form 32 bits) flows to add2.
6. The adder adds, yielding 5000 (in binary), which flows to the RF's wd input. The wa input has $t6's encoding of 01110 already waiting.
7. All those Execute actions happened during one clock cycle. On the next rising clock, 5000 will be written to $t6, finishing the addi instruction. Also, 4 will be written into PC, causing lw's actions to commence.

One should examine the Execute state's control logic actions for each fetched instruction's opcode (recalling that all other control outputs are set with 0), and consider how those actions cause items to flow through the components.

PARTICIPATION ACTIVITY 5.7.5: Executing addi on the MIPSzy processor.

Consider the MIPSzy processor above, carrying out the addi instruction at IM[0]. Assume addi's opcode of 001000 has already flowed back to CTRL.

1) CTRL sets output rf_r1e to ___.

- 0
- 1

2) The wires labeled add2 have a mux. CTRL sets that mux's select input to ___.
   - 0
   - 1

3) ADD will add the read register and ir15_0. The result should flow back to RF's write data input wd. Is a mux involved?
   - Yes
   - No

4) ADD will add the read register and ir15_0. The result should be written to RF[ir20_16]. Is a mux involved?
   - Yes
   - No

5) Is DM either written or read by addi's actions?
   - Yes
   - No

6) Is ir5_0 used by the control logic for addi?
   - Yes
   - No

CHALLENGE ACTIVITY 5.7.1: Instruction execution on MIPSzy processor.

459784.3174716.qx3zqy7

5.8 Base MIPSzy + sub

A designer can extend the base MIPSzy to support the sub (subtract) instruction. The designer can include the sub instruction in the behavioral description. As a reminder, the add and sub instructions have the same opcode bits ir31_26 of 000000 but have differing function bits ir5_0 of 100000 (add) and 100010 (sub).

Table 5.8.1: Reminder: MIPSzy machine instructions for add and sub.

Assembly	Machine
`add $t0, $t1, $t2`	`000000 01001 01010 01000 00000 100000`
`sub $t0, $t1, $t2`	`000000 01001 01010 01000 00000 100010`

To support the behavioral description's actions in the processor's circuit, the designer can replace ADD with an ADD/SUB component that performs addition when the component's control input sub is 0, and subtraction when sub is 1. Then CTRL sets the component's sub input appropriately for each of the add and sub instructions.

PARTICIPATION ACTIVITY 5.8.1: Extending the base MIPSzy processor to support the subtraction instruction.

Animation captions:

1. A designer can introduce the sub instruction to the base MIPSzy's behavioral description. If opcode = 000000 and function field = 100010, a subtraction is performed.
2. In the processor circuit, the designer can replace the adder component with an adder/subtractor component. That component has a control signal that CTRL must set to perform add or subtract.
3. The control logic actions for sub are the same as add, but with the add_sub control signal set with 1.

PARTICIPATION ACTIVITY 5.8.2: Extending the base MIPSzy with the sub instruction.

1) In the base MIPSzy from a previous section, the add instruction's control logic actions set the add_sub control signal with 0.
   - ○ True
   - ○ False

2) In the base MIPSzy extended for sub, the add instruction's control logic actions set the add_sub control signal with 0.
   - ○ True
   - ○ False

3) In the base MIPSzy extended for sub, the sub instruction's control logic

actions set the add_sub control signal with 1.

- True
- False

CHALLENGE ACTIVITY 5.8.1: Subtraction extension in MIPSzy.

459784.3174716.qx3zqy7

5.9 Base MIPSzy + j / jal

j (jump)

A designer can extend the base MIPSzy to support the j (jump) instruction by first modifying the behavioral description to detect opcode 000010, with the actions being to load PC with ir25_0 appropriately modified into a 32-bit address (described in an earlier section on jump/branch immediates).

The designer can then modify the processor circuit to carry out those actions, by inserting a mux in front of the PC and appropriately controlling that mux, with the new value coming from ir25_0 appropriately modified.

PARTICIPATION ACTIVITY 5.9.1: Extending the base MIPSzy processor to support the jump instruction.

Animation captions:

1. A designer can introduce the j instruction to base MIPSzy's behavioral description. If opcode = 000010, PC gets loaded with ir25_0 (prepended with PC's high 4 bits, and appended with 00).
2. The designer can modify the processor circuit to support such a loading of PC, using concatenating wires, and inserting a mux.
3. The designer can modify the control logic's actions to pass the jump address through the PC's mux for a jump, by setting pc_s = 1. (pc_s will be 0 for other instructions, passing PC + 4 instead).

PARTICIPATION ACTIVITY 5.9.2: Extending the base MIPSzy processor to support the jump instruction.

1) In the base MIPSzy processor, CTRL sets pc_s with ___.
   - 0
   - 1
   - Not applicable

2) In the base MIPSzy processor extended for jump, for most instructions, CTRL sets pc_s with ___.

- 0
- 1
- PC + 4

3) In the base MIPSzy processor extended for jump, for the jump instruction, CTRL sets pc_s with ___.

- 0
- 1

4) In the base MIPSzy processor extended for jump, how many bits is the jump target address that gets passed through the PC's mux?

- 26
- 28
- 32

jal (jump and link)

The jal instruction also jumps to a target address like the j instruction, but also writes PC + 4 to the special $ra register, which happens to be at register file location 31. The animation below shows the update to the behavioral description. The animation also shows how larger muxes are needed in front of the RF's wd and wa inputs to pass PC + 4 and 31 respectively. The animation shows the control logic actions to carry out jal on the revised processor circuit.

PARTICIPATION ACTIVITY 5.9.3: Extending MIPSzy to support the jal instruction as well as the j instruction.

Animation captions:

1. The jal instruction is similar to j, except jal also writes PC + 4 to register $ra (which is at location 31 in the RF). Note that the description has RF[31] = PC because PC was already incremented by 4.
2. Larger muxes for RF's wd and wa inputs are needed, to pass PC+4 to wd, and value 31 to wa. The single select control signal for the wd mux is replaced by 2-bit signal rf_wd_s1s0. Likewise for wa.
3. The control logic actions configure the RF and its muxes to write PC + 4 to RF[31] (register $ra), and like the j instruction, configure the PC mux to pass the jump instruction's target address via pc_s = 1.

PARTICIPATION ACTIVITY 5.9.4: Extending the base MIPSzy processor to support the jal instruction.

Consider the base MIPSzy extended to support jal.

1) An add instruction would set rf_wd_s1s0 with ___.

- 1
- 01

2) A jal instruction writes to the register at RF location ___.

- 0
- 31

3) The circuit for loading the PC is _____ for the j and jal instructions.

- the same
- different

CHALLENGE ACTIVITY 5.9.1: j and jal extension in MIPSzy.

459784.3174716.qx3zqy7

5.10 Base MIPSzy + beq/bne

A designer can extend the base MIPSzy to support the beq (branch on equal) instruction. The designer might first modify the behavioral description to detect opcode 000100, with the actions being to load PC with the target address if the instruction's two RF registers have equal values.

The designer can then modify the processor circuit to carry out those actions. The designer would extend the ADD component to have an output signal indicating whether the component's two data inputs are equal. Because the component now does more than just add, the designer might name the component ALU. An ***ALU*** (***arithmetic-logic unit***) is a combinational component that performs the various arithmetic and logic operations needed by a processor, like add, subtract, compare, AND, etc. This ALU only does add and compare-for-equality.

The designer would also add a mux in front of the PC, with the new value coming from ir15_0 with 00 appended and summed with PC + 4. Finally, the designer would update the control logic to feed the two RF registers to the ALU, then control the PC mux based on the value of alu_equal.

PARTICIPATION ACTIVITY 5.10.1: Extending the base MIPSzy for the beq instruction.

Animation captions:

1. A designer can add behavior that detects beq's opcode of 000100. The actions read the two RF registers specified by ir25_21 and ir20_16, and if the register values are equal, update PC with the branch address.
2. The branch address is computed by appending 00 to the offset in ir15_0, then adding with PC, which was already incremented by 4. Recall that the offset was defined taking into account that increment by 4.
3. To support beq's behavioral actions, the designer may extend the ADD component into an ALU that also does comparison for equality.
4. The designer also inserts a component to add PC and the offset, and a mux to pass that target address or the regular next address (PC + 4) to the PC.
5. Finally, the designer creates control logic actions that read the RF registers, then sets pc_s with 1 if alu_eq is 1 to load the branch address, else setting pc_s with 0 to just load PC + 4.

The bne (branch on not equal) is nearly identical, but with opcode 000101 (rather than 000100), and with comparison alu_eq == 0 (rather than 1). No changes to the processor circuit are required; only the behavioral and control logic actions change.

PARTICIPATION ACTIVITY 5.10.2: Extending the base MIPSzy to support beq and bne instructions.

Consider the base MIPSzy extended to support beq and bne.

1) In the processor circuit, if the two register values being input to the ALU are 27 and 13, the ALU's eq output is ___.
   - 0
   - 1

2) In the processor circuit, the beq (and bne) instruction forms the offset by appending's two 0's to ir15_0, and sign extending to ___ bits total.
   - 18
   - 32

3) In the processor circuit, the offset is added with ___.
   - PC + 4
   - the ALU's output

4) If the two registers of a *beq* instruction have equal values, then alu_eq coming into the control logic will be 1. The control logic will thus set pc_s with ___.
   - 0
   - 1

5) If the two registers of a *bne* instruction have equal values, then alu_eq coming into the control logic will be 1. The control logic will thus set pc_s with ___.
   - 0
   - 1

CHALLENGE ACTIVITY 5.10.1: beq/bne extension in MIPSzy.

459784.3174716.qx3zqy7

5.11 Base MIPSzy + slt

A designer can extend the base MIPSzy to support the slt instruction. The designer might first modify the behavioral description to detect opcode 000000 and function code 101010, with the actions being to set the first operand register with 1 if second register is less than the third, else setting the first register with 0.

The designer can then modify the processor circuit to carry out those actions. The designer can extend the ADD component to have an input signal indicating that the component should perform a less-than comparison rather than an addition, outputting 1 if input A is less than input B, else outputting 0 (the 1 and 0 are actually 32 bits: 00..01 and 00..00). Because the component now does more than just add, the designer might name the component ALU. An ***ALU*** (***arithmetic-logic unit***) is a combinational component that performs the various arithmetic and logic operations needed by a processor, like add, subtract, compare, AND, etc. This ALU only does add and compare-for-less.

Finally, the designer would update the control logic to feed the two RF registers to the ALU and set alu_less = 1, and write the result back to the RF (the paths for those reads and write already exist to support the very similar add instruction).

PARTICIPATION ACTIVITY 5.11.1: Creating a circuit of components that supports the MIPSzy behavioral description's actions.

Animation captions:

1. New behavior detects slt's opcode 000000 and function code 101010. The actions read two RF regs at ir25_21 and ir20_16, and set reg ir15_11 with the result of a less-than comparison (yielding 1 or 0).
2. To support slt's behavioral actions, the designer may extend the ADD component into an ALU with an input that when 1, causes a less-than comparison rather than an addition, thus outputting 1 if add1 < add2, else 0.
3. Finally, the designer updates the control logic actions to read and write the registers (just like for an add instruction), but sets alu_less with 1. (The add and addi instructions would set alu_less with 0.)

PARTICIPATION ACTIVITY 5.11.2: Extending the base MIPSzy to support the slt instruction.

1) If add1 is less than add2, what does the ALU output? Type one digit.

Check **Show answer**

2) The control logic actions for slt are nearly identical to those for add. Type the statement that was added for slt.

Check **Show answer**

3) If add1 is not less than add2 (being equal or greater), a 0 will be written into the register specified by which ir bits? Type answer as: ir5_0

Check **Show answer**

CHALLENGE ACTIVITY 5.11.1: slt extension in MIPSzy.

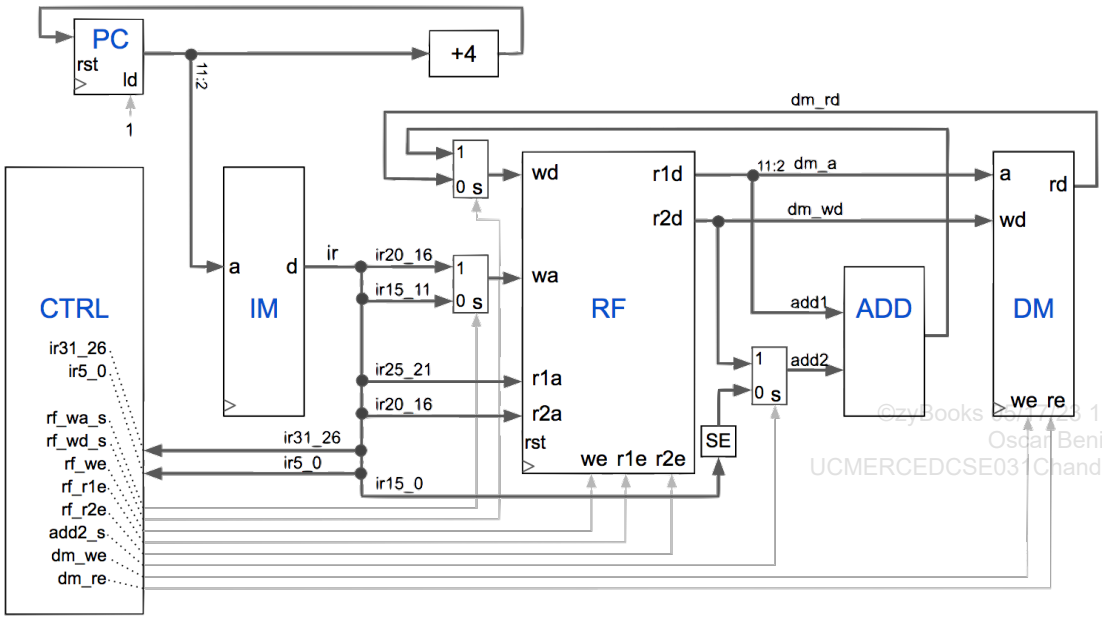
459784.3174716.qx3zqy7

5.12 Base MIPSzy: Verilog

Top-level structure

An earlier section introduced the base MIPSzy's processor design. The base MIPSzy only supports the lw, sw, addi, and add instructions. The base MIPSzy is kept simple so the design's main concepts aren't lost in details of supporting other instructions. The design is shown again below. The design connects several components like CTRL, IM, RF, ADD, DM, PC, SE, +4, and some muxes.

Figure 5.12.1: Base MIPSzy processor design.



A ***hardware description language*** (***HDL***) is a language for describing digital circuit designs. ***Verilog*** is a popular

hardware description language. An HDL can describe a design's structure, meaning the connection of components. Below is Verilog describing MIPSzy's top-level structure. The design is a "module" with inputs clk and rst (clock and reset). The module lists numerous wires, each with a name, like rf_wd_s (one bit) and ir (32 bits). The module introduces several components (known as "instantiating" components) like IM, PC, and RF_wd_mux, each an instance of a particular component type like mem_1024x32, register_32, and mux2x1_32 (respectively). Each component instantiation also lists the wires that connect to that component's inputs and outputs, as in PC (clk, rst, 1, pc_inc4, pc_out). Each such component will be defined later in the Verilog; this module merely instantiates and connects those "top-level" components.

©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

Figure 5.12.2: Verilog for MIPSzy's top-level structure.

```
`timescale 1ns / 1ps
module MIPSzy(clk, rst);
   input clk, rst;

   // Ctrl wires
   wire rf_wd_s, rf_wa_s, rf_we, rf_r1e, rf_r2e;
   wire add2_s;
   wire dm_we, dm_re;

   // PC wires
   wire [31:0] pc_out, pc_inc4;

   // IM/IR wires
   wire [31:0] ir;
   wire [5:0] ir31_26, ir5_0;
   wire [4:0] ir20_16, ir15_11, ir25_21;
   wire [15:0] ir15_0;

   // SW wires
   wire [31:0] ir15_0_se;

   // RF wires
   wire [4:0] rf_wa;
   wire [31:0] rf_wd;
   wire [31:0] rf_r1d, rf_r2d;

   // Adder wires
   wire [31:0] add2, add1, add_sum;

   // DM wires
   wire [31:0] dm_a;
   wire [31:0] dm_wd;
   wire [31:0] dm_rd;

   // internal connections
   assign dm_a = rf_r1d;
   assign dm_wd = rf_r2d;
   assign add1 = rf_r1d;

   assign ir31_26 = ir[31:26];
   assign ir5_0 = ir[5:0];
   assign ir20_16 = ir[20:16];
   assign ir15_11 = ir[15:11];
   assign ir25_21 = ir[25:21];
   assign ir15_0 = ir[15:0];

   // Component instantiations
   mipszy_ctrl    CTRL(ir31_26, ir5_0,
                       rf_wd_s, rf_wa_s, rf_we, rf_r1e,
rf_r2e,
                       add2_s, dm_we, dm_re);
   register_32    PC(clk, rst, 1, pc_inc4, pc_out);
   inc4_32        PC_inc(pc_out, pc_inc4);
   mux2x1_32      RF_wd_mux(add_sum, dm_rd, rf_wd_s,
rf_wd);
   mux2x1_5       RF_wa_mux(ir20_16, ir15_11, rf_wa_s,
rf_wa);
   regfile_32x32  RF(clk, rst,
                     ir25_21, rf_r1d, rf_r1e,
                     ir20_16, rf_r2d, rf_r2e,
                     rf_wa, rf_wd, rf_we);
   signext_16_32  SE(ir15_0, ir15_0_se);
   mux2x1_32      ADD_mux(rf_r2d, ir15_0_se, add2_s, add2);
```

```
    adder_32       ADD(add1, add2, add_sum);
    mem_1024x32    IM(clk, pc_out[11:2], ir, 1, 0, 0, 0);
    mem_1024x32    DM(clk, dm_a[11:2], dm_rd, dm_re,
                      dm_a[11:2], dm_wd, dm_we);
endmodule
```

PARTICIPATION ACTIVITY 5.12.1: Base MIPSzy's top-level structure in Verilog.

1) How many bits is the dm_we wire?
   - 1
   - 32

2) How many bits is the ir wire?
   - 1
   - 32

3) ir31_26 is defined using an _____ statement.
   - assign
   - component instantiation

4) The PC_inc component instance connects to wire pc_out and wire ____.
   - pc_inc4
   - clk

5) Wire pc_inc4 connects component instance PC_inc to component instance ____.
   - pc_out
   - PC

6) How many 32-bit 2x1 muxes are instantiated?
   - 1
   - 2

CTRL: Control logic

MIPSzy's CTRL component carries out the control logic actions for a given instruction. In the Verilog above, CTRL was instantiated as a component of type mipszy_ctrl, which can be described in Verilog as a module having combinational logic behavior, shown below. CTRL's inputs are ir31_26 and ir5_0, which are the opcode and function fields of an instruction. CTRL's outputs are numerous single-bit signals that control the other components in the design, like dm_we which enables writing to the DM component, or add2_s which controls the mux in front of the ADD component's second input. The "always" represents a task, which executes if any input changes (indicated by "@*").

The task determines the instruction (lw, sw, addi, or add) from the input values, and assigns the output control signals with values that carry out that instruction.

©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

Figure 5.12.3: Verilog for MIPSzy's control logic component CTRL.

```
`timescale 1ns / 1ps
module mipszy_ctrl(ir31_26, ir5_0,
                   rf_wd_s, rf_wa_s, rf_we, rf_r1e,
rf_r2e,
                   add2_s,
                   dm_we, dm_re);

   input [5:0] ir31_26, ir5_0;
   output reg rf_wd_s, rf_wa_s, rf_we, rf_r1e, rf_r2e;
   output reg add2_s;
   output reg dm_we, dm_re;

   always @* begin
      if (ir31_26 == 'b100011) begin // lw
         rf_wd_s = 0;
         rf_wa_s = 1;
         rf_we = 1;
         rf_r1e = 1;
         rf_r2e = 0;
         add2_s = 0;
         dm_we = 0;
         dm_re = 1;
      end
      else if (ir31_26 == 'b101011) begin // sw
         rf_wd_s = 0;
         rf_wa_s = 0;
         rf_we = 0;
         rf_r1e = 1;
         rf_r2e = 1;
         add2_s = 0;
         dm_we = 1;
         dm_re = 0;
      end
      else if (ir31_26 == 'b001000) begin // addi
         rf_wd_s = 1;
         rf_wa_s = 1;
         rf_we = 1;
         rf_r1e = 1;
         rf_r2e = 0;
         add2_s = 0;
         dm_we = 0;
         dm_re = 0;
      end
      else if (ir31_26 == 'b000000 &amp;&amp;
               ir5_0   == 'b100000) begin // add
         rf_wd_s = 1;
         rf_wa_s = 0;
         rf_we = 1;
         rf_r1e = 1;
         rf_r2e = 1;
         add2_s = 1;
         dm_we = 0;
         dm_re = 0;
      end
      else begin
         rf_wd_s = 0;
         rf_wa_s = 0;
         rf_we = 0;
         rf_r1e = 0;
         rf_r2e = 0;
         add2_s = 0;
         dm_we = 0;
         dm_re = 0;
```

```
            end
        end
    endmodule
```

PARTICIPATION ACTIVITY 5.12.2: Base MIPSzy's control logic.

1) Is dm_we an input or output of the control logic?
   - Input
   - Output

2) How does the description detect an addi instruction?
   - (ir31_26 == 'b100011)
   - (ir31_26 == 'b001000)
   - (ir31_26 == 'b000000 && ir5_0 == 'b100000)

3) What does the description do if the opcode isn't recognized?
   - Nothing
   - Assigns all outputs with 0's
   - Exit

Register, memory, and register file

Below are Verilog descriptions for a 32-bit register (used for PC), a 1024x32 memory (used for IM and DM), and a 32x32 register file (used for RF) component.

The register is only updating on rising clock edges (i.e., is "synchronous"). If rst is 1, the register is reset to 0, and otherwise is loaded with input D.

The memory is described using an array of 1024 registers ([0:1023]). The memory has a task to handle writes synchronously (executes only on a rising clock edge), and another task to handle reads asynchronously (executes if any input changes, via "@*").

The register file is described using an array of 32 registers ([0:31]). The register file has a task to handle writes synchronously (executes only on a rising clock edge), and another task to handle reads asynchronously (executes if any input changes, via "@*"). The read task handles both read ports.

Figure 5.12.4: Verilog for 32-bit register used for PC.

```
`timescale 1ns / 1ps
module register_32(clk, rst, ld, D,
Q);
   input clk, rst;
   input ld;
   input [31:0] D;
   output reg [31:0] Q;

   always @(posedge clk) begin
      if (rst) begin
         Q = 0;
      end
      else if (ld) begin
         Q = D;
      end
   end
endmodule
```


Figure 5.12.5: Verilog for 1024x32 memory used for instruction and data memories.

```
`timescale 1ns / 1ps
module mem_1024x32(clk, ra, rd, re, wa, wd, we);
   input clk;
   input [9:0] ra, wa;
   input re, we;
   output reg [31:0] rd;
   input [31:0] wd;

   reg [31:0] memory [0:1023];

   always @(posedge clk) begin
     if (we) begin
         memory[wa] = wd;
      end
   end

   always @* begin
      if (re) begin
         rd = memory[ra];
      end
      else begin
         rd = 0;
      end
   end
endmodule
```

Figure 5.12.6: Verilog for 32x32 register file with 2 read ports and 1 write port.

```
`timescale 1ns / 1ps
module regfile_32x32(clk, rst,
                     r1a, r1d, r1e,
                     r2a, r2d, r2e,
                     wa, wd, we);
   input clk, rst;
   input [4:0] r1a, r2a, wa;
   input r1e, r2e, we;
   output reg [31:0] r1d, r2d;
   input [31:0] wd;

   integer i;
   reg [31:0] registers [0:31];

   // Write procedure
   always @(posedge clk) begin
      if (rst) begin
         for (i = 0; i < 32; i = i+1)
begin
            registers[i] = 0;
         end
      end
      else if (we) begin
         registers[wa] = wd;
      end
   end

   // Read ports procedure
   always @* begin
      // Read port 1
      if (r1e) begin
         r1d = registers[r1a];
      end
      else begin
         r1d = 0;
      end

      // Read port 2
      if (r2e) begin
         r2d = registers[r2a];
      end
      else begin
         r2d = 0;
      end
   end
endmodule
```


PARTICIPATION ACTIVITY 5.12.3: Base MIPSzy's storage components: PC, memory, and RF.

1) For register_32 , what happens to the register's output value Q if rst changes from 0 to 1 and clk stays at 0?

- ○ Q = 0
- ○ Nothing

2) How many component instances in the base MIPSzy's top-level structure are a 32-bit register?
   - 1
   - 3

3) For mem_1024x32, on a positive clock edge, what value of input we causes the memory's array to be updated?
   - 0
   - 1

4) For mem_1024x32, how many bits is input wa?
   - 10
   - 32

5) For mem_1024x32, is the read synchronous?
   - Yes
   - No

6) For regfile_32x32, does the task handle the case of both r1e and r2e being 1's?
   - Yes
   - No

7) For regfile_32x32, what is output on read port 1 if r1e is 0?
   - 0
   - Nothing

Other combinational components

Below are Verilog descriptions for the other combinational components in the base MIPSzy's design.

Figure 5.12.7: Verilog for 32-bit adder.

```
`timescale 1ns / 1ps
module adder_32(A, B,
S);
   input [31:0] A, B;
   output reg [31:0] S;

   always @(A, B) begin
      S = A + B;
   end
endmodule
```


Figure 5.12.8: Verilog for 32-bit incrementer that increments by 4.

```
`timescale 1ns / 1ps
module inc4_32(A, S);
   input [31:0] A;
   output reg [31:0]
S;

   always @(A) begin
      S = A + 4;
   end
endmodule
```

Figure 5.12.9: Verilog for 32-bit 2x1 mux.

```
`timescale 1ns / 1ps
module mux2x1_32(I1, I0, s,
D);
   input [31:0] I1, I0;
   input s;
   output reg [31:0] D;

   always @(I1, I0, s)
   begin
      if (!s) begin
         D = I0;
      end
      else begin
         D = I1;
      end
   end
endmodule
```

Figure 5.12.10: Verilog for 5-bit 2x1 mux.

```
`timescale 1ns / 1ps
module mux2x1_5(I1, I0, s,
D);
   input [4:0] I1, I0;
   input s;
   output reg [4:0] D;

   always @(I1, I0, s)
   begin
      if (!s) begin
         D = I0;
      end
      else begin
         D = I1;
      end
   end
endmodule
```


Figure 5.12.11: Verilog for 16-bit to 32-bit sign extension.

```
`timescale 1ns / 1ps
module signext_16_32(I,
O);
   input signed [15:0] I;
   output reg [31:0] O;

   always @(I) begin
      O = I;
   end
endmodule
```

PARTICIPATION ACTIVITY 5.12.4: Base MIPSzy's other combinational components.

Match the combinational component with the Verilog behavior.

If unable to drag and drop, refresh the page.

mux2x1_32 **adder_32** **inc4_32** **signext_16_32**

S = A + B;

S = A + 4;

if (!s) begin D = I0; end else begin D = I1;

O = I;

Reset

Testbench

A designer can test the base MIPSzy's Verilog using a testbench. The testbench instantiates the base MIPSzy's top-level module MIPSzy, naming the instance MIPSzy_0. The testbench initializes that module's IM component's first four words with four machine instructions, and one DM word (at address 5000) with the value 10 (an "initial" task runs only once, at the start of the Verilog's execution). The testbench uses another "initial" task to assign the rst input with 1 for one clock cycle. The testbench has an "always" task that simply pulses the clk signal repeatedly. That testbench will cause the four instructions in IM to execute on the MIPSzy's design. A designer might set up the Verilog simulator to view DM's values, to see if DM gets updated as expected (the 10 should be doubled to 20).

Figure 5.12.12: Simple testbench for MIPSzy processor.

```
`timescale 1ns / 1ps
module MIPSzy_TB();
   reg clk_tb, rst_tb;

   localparam CLK_PERIOD = 20;

   MIPSzy MIPSzy_0(clk_tb, rst_tb);

   initial begin
      // initialize instruction memory
      MIPSzy_0.IM.memory[0] = 'b00100000000011100001001110001000; // addi $t6,
$zero, 5000
      MIPSzy_0.IM.memory[1] = 'b10001101110010000000000000000000; // lw $t0, 0($t6)
# Load from DM[5000]
      MIPSzy_0.IM.memory[2] = 'b00000001000010000100100000100000; // add $t1, $t0,
$t0 # Double the values
      MIPSzy_0.IM.memory[3] = 'b10101101110010010000000000000000; // sw $t1, 0($t6)
# Store to DM[5000]

      MIPSzy_0.DM.memory[(5000-4096)/4] = 10;
   end


   // Generate clock
   always begin
      clk_tb = 0;
      #(CLK_PERIOD / 2);
      clk_tb = 1;
      #(CLK_PERIOD / 2);
   end

   initial begin
      rst_tb = 1;
      #CLK_PERIOD rst_tb = 0;
   end

endmodule
```


PARTICIPATION ACTIVITY 5.12.5: Base MIPSzy's testbench.

1) The number of clock cycles that the testbench should execute is ____.

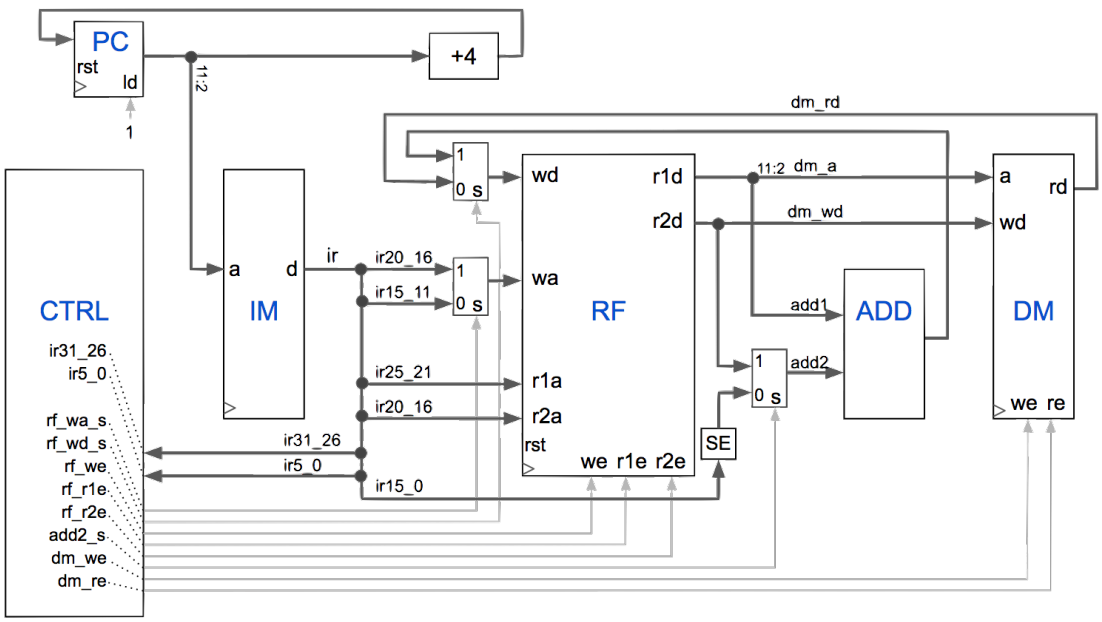
- 10
- unlimited

5.13 Base MIPSzy: VHDL

Top-level structure

An earlier section introduced the base MIPSzy's processor design. The base MIPSzy only supports the lw, sw, addi, and add instructions. The base MIPSzy is kept simple so the design's main concepts aren't lost in details of supporting other instructions. The design is shown again below. The design connects several components like CTRL, IM, RF, ADD, DM, PC, SE, +4, and some muxes.

Figure 5.13.1: Base MIPSzy processor design.



A ***hardware description language*** (***HDL***) is a language for describing digital circuit designs. ***VHDL*** (Very High Speed Integrated Circuit Hardware Description Language) is a popular hardware description language. An HDL can describe a design's structure, meaning the connection of components. Below is the VHDL describing MIPSzy's top-level structure. The design has an "entity" with inputs clk and rst (clock and reset). The design lists numerous signals, each with a name, like rf_wd_s (one bit) and ir (32 bits). The design has an architecture that introduces several components (known as "instantiating" components) like IM, PC, and RF_wd_mux, each an instance of a particular component type like mem_1024x32, register_32, and mux2x1_32 (respectively). Each component instantiation also lists the signals that connect to that component's inputs and outputs, as in PC (clk, rst, 1, pc_inc4, pc_out). Each such component will be defined later in the VHDL. This design merely instantiates and connects those "top-level" components.

Figure 5.13.2: VHDL for MIPSzy's top-level structure.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.NUMERIC_STD.all;

entity MIPSzy is
   port (clk :  in std_logic;
         rst :  in std_logic);
end entity;

architecture Behavior of MIPSzy is
   -- CTRL signals
   signal rf_wd_s, rf_wa_s, rf_we, rf_r1e, rf_r2e : std_logic;
   signal add2_s : std_logic;
   signal dm_we, dm_re : std_logic;
   -- PC signals
   signal pc_out, pc_inc4 : std_logic_vector(31 downto 0);
   -- IM/IR signals
   signal ir : std_logic_vector(31 downto 0);
   signal ir31_26, ir5_0 : std_logic_vector(5 downto 0);
   signal ir20_16, ir15_11, ir25_21 : std_logic_vector(4 downto 0);
   signal ir15_0 : std_logic_vector(15 downto 0);
   -- SW signals
   signal ir15_0_se : std_logic_vector(31 downto 0);
   -- RF signals
   signal rf_wa : std_logic_vector(4 downto 0);
   signal rf_wd : std_logic_vector(31 downto 0);
   signal rf_r1d, rf_r2d : std_logic_vector(31 downto 0);
   -- Adder signals
   signal add2, add1, add_sum : std_logic_vector(31 downto 0);
   -- DM signals
   signal dm_a : std_logic_vector(31 downto 0);
   signal dm_wd : std_logic_vector(31 downto 0);
   signal dm_rd : std_logic_vector(31 downto 0);

   component MIPSzy_ctrl is
      port (
         ir31_26 : in std_logic_vector(5 downto 0);
         ir5_0   : in std_logic_vector(5 downto 0);
         rf_wd_s : out std_logic;
         rf_wa_s : out std_logic;
         rf_we   : out std_logic;
         rf_r1e  : out std_logic;
         rf_r2e  : out std_logic;
         add2_s  : out std_logic;
         dm_we   : out std_logic;
         dm_re   : out std_logic
        );
    end component;

    component register_32 is
       port (
          clk : in std_logic;
          rst : in std_logic;
          ld  : in std_logic;
          D   : in std_logic_vector(31 downto 0);
          Q   : out std_logic_vector(31 downto 0)
       );
    end component;

    component inc4_32 is
       port (
          A : in std_logic_vector(31 downto 0);
```

```
            S : out std_logic_vector(31 downto 0)
        );
    end component;

    component mux2x1_32 is
        port (
            I1 : in std_logic_vector(31 downto 0);
            I0 : in std_logic_vector(31 downto 0);
            s  : in std_logic;
            D  : out std_logic_vector(31 downto 0)
        );
    end component;

    component mux2x1_5 is
        port (
            I1 : in std_logic_vector(4 downto 0);
            I0 : in std_logic_vector(4 downto 0);
            s  : in std_logic;
            D  : out std_logic_vector(4 downto 0)
        );
    end component;

    component regfile_32x32 is
        port (
            clk : in std_logic;
            rst : in std_logic;
            r1a : in std_logic_vector(4 downto 0);
            r1d : out std_logic_vector(31 downto 0);
            r1e : in std_logic;
            r2a : in std_logic_vector(4 downto 0);
            r2d : out std_logic_vector(31 downto 0);
            r2e : in std_logic;
            wa  : in std_logic_vector(4 downto 0);
            wd  : in std_logic_vector(31 downto 0);
            we  : in std_logic
        );
    end component;

    component signext_16_32 is
        port (
            I : in std_logic_vector(15 downto 0);
            O : out std_logic_vector(31 downto 0)
        );
    end component;

    component adder_32 is
        port (
            A : in std_logic_vector(31 downto 0);
            B : in std_logic_vector(31 downto 0);
            S : out std_logic_vector(31 downto 0)
        );
    end component;

    component mem_1024x32 is
        port (
            clk : in std_logic;
            ra  : in std_logic_vector(9 downto 0);
            rd  : out std_logic_vector(31 downto 0);
            re  : in std_logic;
            wa  : in std_logic_vector(9 downto 0);
            wd  : in std_logic_vector(31 downto 0);
            we  : in std_logic
         );
    end component;

begin
    -- internal connections
    dm_a <= rf_r1d;
    dm_wd <= rf_r2d;
    add1 <= rf_r1d;
```


©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

```
    add1 <= rf_r1d;
    ir31_26 <= ir(31 downto 26);
    ir5_0 <= ir(5 downto 0);
    ir20_16 <= ir(20 downto 16);
    ir15_11 <= ir(15 downto 11);
    ir25_21 <= ir(25 downto 21);
    ir15_0 <= ir(15 downto 0);

    -- Component instantiations
    CTRL : MIPSzy_ctrl port map (ir31_26, ir5_0, rf_wd_s, rf_wa_s,
                                 rf_we, rf_r1e, rf_r2e, add2_s,
                                 dm_we, dm_re);
    PC : register_32 port map (clk, rst, '1', pc_inc4, pc_out);
    PC_inc : inc4_32 port map (pc_out, pc_inc4);
    RF_wd_mux : mux2x1_32 port map (add_sum, dm_rd, rf_wd_s, rf_wd);
    RF_wa_mux : mux2x1_5 port map (ir20_16, ir15_11, rf_wa_s, rf_wa);
    RF : regfile_32x32 port map (clk, rst, ir25_21, rf_r1d, rf_r1e,
                                 ir20_16, rf_r2d, rf_r2e, rf_wa,
                                 rf_wd, rf_we);
    SE : signext_16_32 port map (ir15_0, ir15_0_se);
    ADD_mux : mux2x1_32 port map (rf_r2d, ir15_0_se, add2_s, add2);
    ADD : adder_32 port map (add1, add2, add_sum);
    IM : mem_1024x32 port map (clk => clk, ra => pc_out(11 downto 2),
                               wa => (others => '0'), re => '1',
                               we => '0', rd => ir, wd => (others =>
'0'));
    DM : mem_1024x32 port map (clk => clk, ra => dm_a(11 downto 2),
                               wa => dm_a(11 downto 2), re => dm_re,
                               we => dm_we, rd => dm_rd, wd => dm_wd);
end Behavior;
```


PARTICIPATION ACTIVITY 5.13.1: Base MIPSzy's top-level structure in VHDL.

1) How many bits is the dm_we signal?
   - 1
   - 32

2) How many bits is the ir signal?
   - 1
   - 32

3) ir31_26 is defined using an ____ statement.
   - signal assignment
   - component instantiation

4) The PC_inc component instance connects to signal pc_out and signal ____.
   - pc_inc4
   - clk

5) Signal pc_inc4 connects component instance PC_inc to component instance

___.

- pc_out
- PC

6) How many 32-bit 2x1 muxes are instantiated?

- 1
- 2

CTRL: Control logic

MIPSzy's CTRL component carries out the control logic actions for a given instruction. In the VHDL above, CTRL was instantiated as a component of type MIPSzy_ctrl, which can be described in VHDL as a module having combinational logic behavior, shown below. CTRL's inputs are ir31_26 and ir5_0, which are the opcode and function fields of an instruction. CTRL's outputs are numerous single-bit signals that control the other components in the design, like dm_we which enables writing to the DM component, or add2_s which controls the mux in front of the ADD component's second input. The process executes if either ir31_26 or ir5_0 input changes. The process determines the instruction (lw, sw, addi, or add) from the input values, and assigns the output control signals with values that carry out that instruction.

©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

Figure 5.13.3: VHDL for MIPSzy's control logic component CTRL.

```
library IEEE;
library work;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_MISC.all;
use IEEE.STD_LOGIC_unsigned.all;
use IEEE.NUMERIC_STD.all;

entity MIPSzy_ctrl is
   port (ir31_26, ir5_0 : in std_logic_vector(5 downto 0);
         rf_wd_s, rf_wa_s, rf_we, rf_r1e, rf_r2e : out std_logic;
         add2_s : out std_logic;
         dm_we, dm_re : out std_logic);
end MIPSzy_ctrl;

architecture Behavior of MIPSzy_ctrl is
begin
   process(ir31_26, ir5_0)
   begin
      if (ir31_26 = "100011") then  -- lw
         rf_wd_s <= '0';
         rf_wa_s <= '1';
         rf_we <= '1';
         rf_r1e <= '1';
         rf_r2e <= '0';
         add2_s <= '0';
         dm_we <= '0';
         dm_re <= '1';
      elsif (ir31_26 = "101011") then -- sw
         rf_wd_s <= '0';
         rf_wa_s <= '0';
         rf_we <= '0';
         rf_r1e <= '1';
         rf_r2e <= '1';
         add2_s <= '0';
         dm_we <= '1';
         dm_re <= '0';
      elsif (ir31_26 = "001000") then -- addi
         rf_wd_s <= '1';
         rf_wa_s <= '1';
         rf_we <= '1';
         rf_r1e <= '1';
         rf_r2e <= '0';
         add2_s <= '0';
         dm_we <= '0';
         dm_re <= '0';
      elsif ((ir31_26 = "000000") and (ir5_0 = "100000")) then --
add
         rf_wd_s <= '1';
         rf_wa_s <= '0';
         rf_we <= '1';
         rf_r1e <= '1';
         rf_r2e <= '1';
         add2_s <= '1';
         dm_we <= '0';
         dm_re <= '0';
      else
         rf_wd_s <= '0';
         rf_wa_s <= '0';
         rf_we <= '0';
         rf_r1e <= '0';
         rf_r2e <= '0';
         add2_s <= '0';
         dm_we <= '0';
```


©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

```
                dm_re <= '0';
            end if;
        end process;
    end Behavior;
```

PARTICIPATION ACTIVITY 5.13.2: Base MIPSzy's control logic.

1) Is dm_we an input or output of the control logic?
   - Input
   - Output

2) How does the description detect an addi instruction?
   - (ir31_26 = "100011")
   - (ir31_26 = "001000")
   - (ir31_26 = "000000" and ir5_0 = "100000")

3) What does the description do if the opcode isn't recognized?
   - Nothing
   - Assigns all outputs with 0's
   - Exit

Register, memory, and register file

Below are VHDL descriptions for a 32-bit register (used for PC), a 1024x32 memory (used for IM and DM), and a 32x32 register file (used for RF) component.

The register is only updated on rising clock edges (i.e., is "synchronous"). If rst is 1, the register is reset to 0, and otherwise is loaded with input D.

The memory is described using an array of 1024 32-bit signals. The memory has a process to handle writes synchronously (executes only on a rising clock edge), and another process to handle reads asynchronously.

The register file is described using an array of 32 registers. The register file has a process to handle writes synchronously (executes only on a rising clock edge), and another process to handle reads asynchronously. The read process handles both read ports.

Figure 5.13.4: VHDL for 32-bit register used for PC.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_MISC.all;
use IEEE.NUMERIC_STD.all;

entity register_32 is
   port(clk, rst : in std_logic;
        ld : in std_logic;
        D  : in std_logic_vector(31 downto
0);
        Q  : out std_logic_vector(31 downto
0));
end register_32;

architecture Behavior of register_32 is
begin
   process(clk)
   begin
      if (rising_edge(clk)) then
         if (rst = '1') then
            Q <= x"00000000";
         elsif ld = '1' then
            Q <= D;
         end if;
      end if;
   end process;
end Behavior;
```


©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

Figure 5.13.5: VHDL for 1024x32 memory used for instruction and data memories.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_unsigned.all;
use IEEE.NUMERIC_STD.all;

entity mem_1024x32 is
   port (clk  : in std_logic;
         wa, ra : in std_logic_vector(9 downto 0);
         re, we : in std_logic;
         rd : out std_logic_vector(31 downto 0);
         wd : in std_logic_vector(31 downto 0)
   );
end mem_1024x32;

architecture description of mem_1024x32 is
   type Memory_Type is array(0 to 1023) of std_logic_vector(31 downto
0);
   signal memory : Memory_Type;
begin
   -- write process
   process(clk)
   begin
      if (rising_edge(clk)) then
         if (we = '1') then
            memory(conv_integer(wa)) <= wd;
         end if;
       end if;
   end process;

   -- read process
   process(ra, re, memory)
   begin
      if (re = '1') then
         rd <= memory(conv_integer(ra));
      else
         rd <= (others => '0');
      end if;
   end process;

end architecture;
```


©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

Figure 5.13.6: VHDL for 32x32 register file with 2 read ports and 1 write port.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_MISC.all;
use IEEE.STD_LOGIC_unsigned.all;
use IEEE.NUMERIC_STD.all;

entity regfile_32x32 is
   port (clk, rst : in std_logic;
         r1a, r2a, wa: in std_logic_vector(4 downto 0);
         r1e, r2e, we : in std_logic;
         r1d, r2d : out std_logic_vector(31 downto 0);
         wd : in std_logic_vector(31 downto 0));
end regfile_32x32;

architecture Behavior of regfile_32x32 is
   type reg_type is array (0 to 31) of std_logic_vector (31 downto
0);
   signal registers: reg_type;
begin
   -- Write process
   process(clk)
   begin
      if (rising_edge(clk)) then
         if (rst = '1') then
            for i in 0 to 31 loop
               registers(i) <= (others => '0');
            end loop;
         elsif (we = '1') then
            registers(to_integer(unsigned(wa))) <= wd;
         end if;
      end if;
   end process;

   -- Read ports process
   process(r1e, r2e, r1a, r2a)
   begin
      -- Read port 1
      if (r1e = '1') then
         r1d <= registers(to_integer(unsigned(r1a)));
      else
         r1d <= (others => '0');
      end if;

      -- Read port 2
      if (r2e = '1') then
         r2d <= registers(to_integer(unsigned(r2a)));
      else
         r2d <= (others => '0');
      end if;
   end process;
end Behavior;
```


©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

PARTICIPATION ACTIVITY 5.13.3: Base MIPSzy's storage components: PC, memory, and RF.

1) For register_32, what happens to the register's output value Q if rst changes

from 0 to 1 and clk stays at 0?

- Q = 0
- Nothing

2) How many component instances in the base MIPSzy's top-level structure are a 32-bit register?

- 1
- 3

3) For mem_1024x32, on a rising clock edge, what value of input we causes the memory's array to be updated?

- 0
- 1

4) For mem_1024x32, how many bits is input wa?

- 10
- 32

5) For mem_1024x32, is the read synchronous?

- Yes
- No

6) For regfile_32x32, does the process handle the case of both r1e and r2e being 1's?

- Yes
- No

7) For regfile_32x32, what is output on read port 1 if r1e is 0?

- 0
- Nothing

Other combinational components

Below are VHDL descriptions for the other combinational components in the base MIPSzy's design.

Figure 5.13.7: VHDL for 32-bit adder.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_unsigned.all;

entity adder_32 is
   port (A, B : in std_logic_vector(31 downto 0);
              S : out std_logic_vector(31 downto
0));
end entity adder_32;

architecture Behavior of adder_32 is
begin
   process(A, B)
   begin
      S <= A + B;
   end process;
end Behavior;
```


Figure 5.13.8: VHDL for 32-bit incrementer that increments by 4.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_unsigned.all;

entity inc4_32 is
   port (A : in std_logic_vector(31 downto
0);
         S : out std_logic_vector(31 downto
0));
end entity inc4_32;

architecture Behavior of inc4_32 is
begin
   process(A)
   begin
      S <= A + 4;
   end process;
end Behavior;
```

Figure 5.13.9: VHDL for 32-bit 2x1 mux.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_unsigned.all;

entity mux2x1_32 is
    port (I0, I1 : in  std_logic_vector (31 downto
0);
          s : in  std_logic;
          D : out std_logic_vector (31 downto 0));
end mux2x1_32;

architecture Behavior of mux2x1_32 is
begin
   process(I1, I0, s)
   begin
      if (s = '0') then
         D <= I0;
      else
         D <= I1;
      end if;
   end process;
end Behavior;
```


Figure 5.13.10: VHDL for 5-bit 2x1 mux.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_unsigned.all;

entity mux2x1_5 is
   port (I0 , I1 : in  std_logic_vector (4 downto
0);
         s : in  std_logic;
         D : out std_logic_vector (4 downto 0));
end mux2x1_5;

architecture Behavior of mux2x1_5 is
begin
   process(I1, I0, s)
   begin
      if (s = '0') then
         D <= I0;
      else
         D <= I1;
      end if;
   end process;
end Behavior;
```

Figure 5.13.11: VHDL for 16-bit to 32-bit sign extension.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_unsigned.all;
use IEEE.NUMERIC_STD.all;

entity signext_16_32 is
   port (I : in std_logic_vector(15 downto 0);
         O : out std_logic_vector(31 downto
0));
end entity signext_16_32;

architecture Behavior of signext_16_32 is
begin
   process(I)
   begin
      O <=
std_logic_vector(resize(signed(I),32));
   end process;
end Behavior;
```


PARTICIPATION ACTIVITY 5.13.4: Base MIPSzy's other combinational components.

Match the combinational component with the VHDL behavior.

If unable to drag and drop, refresh the page.

adder_32 **signext_16_32** **inc4_32** **mux2x1_32**

S = A + B;

S = A + 4;

if (s = '0') then D <= I0; else D <= I1; end if;

O <= std_logic_vector(resize(signed(I),32));

Reset

Testbench

A designer can test the base MIPSzy's VHDL using a testbench. The testbench instantiates the base MIPSzy's top-level module MIPSzy, naming the instance MIPSzy_0. The testbench's first process initializes that module's IM component's first four words with four machine instructions, and one DM word (at address 5000) with the value 10. The `wait;` statement at the end of the process ensures the process only runs once at the start of simulation. The

testbench uses another process to assign the rst input with 1 for one clock cycle. The testbench has a third process that generates the clk signal throughout the simulation. That testbench will cause the four instructions in IM to execute on the MIPSzy's design. A designer might set up the VHDL simulator to view DM's values, to see if DM gets updated as expected (the 10 should be doubled to 20).

©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

Figure 5.13.12: Simple testbench for MIPSzy processor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity MIPSzy_TB is
end MIPSzy_TB;

architecture Behavior of MIPSzy_TB is
   -- Component declaration for the MIPSzy processor
   component MIPSzy
      port (clk : in  std_logic;
            rst : in  std_logic);
   end component MIPSzy;

   signal clk_tb : std_logic := '0';
   signal rst_tb : std_logic := '0';

   -- Clock period definitions
   constant CLK_PERIOD : time := 20 ns;

   -- Type declaration and aliases for initializing instruction and data memories
   type Memory_Type is array(0 to 1023) of std_logic_vector(31 downto 0);
   alias IM is << SIGNAL .MIPSzy_TB.MIPSzy_0.IM.memory : Memory_Type >>;
   alias DM is << SIGNAL .MIPSzy_TB.MIPSzy_0.DM.memory : Memory_Type >>;
begin
   -- Instantiate the for the MIPSzy Processor in VHDL
   MIPSzy_0: MIPSzy port map (clk_tb, rst_tb);

   -- Initialize instruction and data memories
   process begin
      IM(0) <= "00100000000011100001001110001000"; -- addi $t6, $zero, 5000
      IM(1) <= "10001101110010000000000000000000"; -- lw $t0, 0($t6)  # Load from
DM[5000]
      IM(2) <= "00000001000010000100100000100000"; -- add $t1, $t0, $t0 # Double
the values
      IM(3) <= "10101101110010010000000000000000"; -- sw $t1, 0($t6)  # Store to
DM[5000]

      DM((5000-4096)/4) <= conv_std_logic_vector(10, 32);
      wait;
   end process;

   -- Generate clock
   process begin
      clk_tb <= '0';
      wait for CLK_PERIOD/2;
      clk_tb <= '1';
      wait for CLK_PERIOD/2;
   end process;

   process begin
      rst_tb <= '1';
      wait for CLK_PERIOD*10;
      rst_tb <= '0';
      wait;
  end process;
end Behavior;
```

PARTICIPATION ACTIVITY 5.13.5: Base MIPSzy's testbench.

1) The number of clock cycles that the testbench should execute is ____.
   - ○ 10
   - ○ unlimited

©zyBooks 05/17/23 13:49 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023