Design of Digital Circuits (252-0028-00L), Spring 2019

Optional HW 3: Verilog, FSM, and Basic Microarchitectural Design

SOLUTIONS

Instructor: Prof. Onur Mutlu

TAs: Mohammed Alser, Can Firtina, Hasan Hassan, Juan Gomez Luna, Lois Orosa, Giray Yaglikci

Released: Wednesday, March 27, 2019

1 Verilog

Please answer the following four questions about Verilog.

(a) Does the following code result in a sequential circuit or a combinational circuit? Explain why.

```
module concat (input clk, input data_in1, input data_in2,
                                output reg [1:0] data_out);
 always @ (posedge clk, data_in1)
   if (data_in1)
      data_out = {data_in1, data_in2};
   else if (data_in2)
      data_out = {data_in2, data_in1};
endmodule
```

Answer and concise explanation:

Sequential circuit.

Explanation.
This code results in a sequential circuit because `data_in2` is *not* in the sensitivity list, and thus a latch is inferred for `data_out`.

(b) The input `clk` is a clock signal. What is the hexadecimal value of the output `c` right after the third positive edge of `clk` if initially `c = 8'hE3` and `a = 4'd8` and `b = 4'o2` during the entire time?

```
1  module mod1 (input clk, input [3:0] a, input [3:0] b, output reg [7:0] c);
2  always @ (posedge clk)
3    begin
4      c <= {c, &a, |b};
5      c[0] <= ^c[7:6];
6    end
7  endmodule
```

Please answer below. Show your work.

8'hC4.

Explanation.
Cycle 1: c <= {c, &a, |b} $\rightarrow$ c <= {1110_0011, 0, 1} $\rightarrow$ c <= {1000_1101}
c[0] <= ^c[7:6] $\rightarrow$ c[0] <= ^{11} $\rightarrow$ c[0] <= 0
At the first positive edge of clk, $c = 8'b1000\_1100$
Cycle 2: c <= {c, &a, |b} $\rightarrow$ c <= {1000_1100, 0, 1} $\rightarrow$ c <= {0011_0001}
c[0] <= ^c[7:6] $\rightarrow$ c[0] <= ^{10} $\rightarrow$ c[0] <= 1
At the second positive edge of clk, $c = 8'b0011\_0001$
Cycle 3: c <= {c, &a, |b} $\rightarrow$ c <= {0011_0001, 0, 1} $\rightarrow$ c <= {1100_0101}
c[0] <= ^c[7:6] $\rightarrow$ c[0] <= ^{00} $\rightarrow$ c[0] <= 0
At the third positive edge of clk, $c = 8'b1100\_0100 \rightarrow c = 8'hC4$

Note that since the assignments to c are non-blocking, $c[7:6]$ in line 5 is not affected by the assignment to c in line 4 in the same cycle.

(c) Is the following code syntactically correct? If not, please explain the mistake(s) and how to fix it/them.

```
module 1nn3r ( input [3:0] d, input op, output[1:0] s);
  assign s = op ? (d[1:0] - d[3:2]) :
                        (d[3:2] + d[1:0]);
endmodule

module top ( input wire [6:0] instr, input wire op, output reg z);

  reg[1:0] r1, r2;

  1nn3r i0 (.instr(instr[1:0]), .op(instr[7]), .z(r1) );
  1nn3r i1 (.instr(instr[3:2]), .op(instr[0]), .z(r2) );
  assign z = r1 | r2;

endmodule
```

Answer and concise explanation:

The code is not syntactically correct.

Explanation.

- Module names cannot start with a number → '1nn3r' is not a legal module name.
- The output signal 'z' has to be declared as a 'wire' but not 'reg'.
- 'r1' and 'r2' has to be declared as 'wire's.
- The module '1nn3r' does not have ports named 'instr' and 'z'. Those need to be changed to 'd' and 's', respectively.

(d) Does the following code correctly implement a counter that counts from 1 to 11 by increments of 2 (e.g., 1, 3, 5, 7, 9, 11, 1, 3 ...)? If so, say "Correct". If not, correct the code with minimal modification.

```
module odd_counter (clk, count);
  wire clk;
  reg[2:0] count;
  reg[2:0] count_next;

  always@*
  begin
    count_next = count;
    if(count != 11)
      count_next = count_next + 2;
    else
      count_next <= 1;
  end

  always@(posedge clk)
    count <= count_next;
endmodule
```

Answer and concise explanation:

No, the implementation is not correct.

Explanation.
The correct implementation:

```
module odd_counter (input clk, output [3:0] count);
  wire clk;
  reg[3:0] count = 1;
  reg[3:0] count_next;

  always@* begin
    count_next = count;
    if(count != 11)
      count_next += 2;
    else
      count_next = 1;
  end

  always@(posedge clk)
    count <= count_next;
endmodule
```

(e) Does the following code correctly instantiate a 4-bit adder? If so, say "Correct". If not, correct the code with minimal modification.

```
module adder(input a, input b, input c, output sum, output carry);
assign sum = a ^ b ^ c;
assign carry = (a&b) | (b&c) | (c&a);
endmodule


module adder_4bits(input [3:0] a, input [3:0] b, output [3:0] sum, carry);
wire [2:0]s;

adder u0 (a[0],b[0],1'b0,sum[0],s[0]);
adder u1 (a[1],s[0],b[1],sum[1],s[1]);
adder u2 (a[2],s[1],b[2],sum[2],s[2]);
adder u3 (a[3],s[2],b[3],sum[3],carry);
endmodule
```

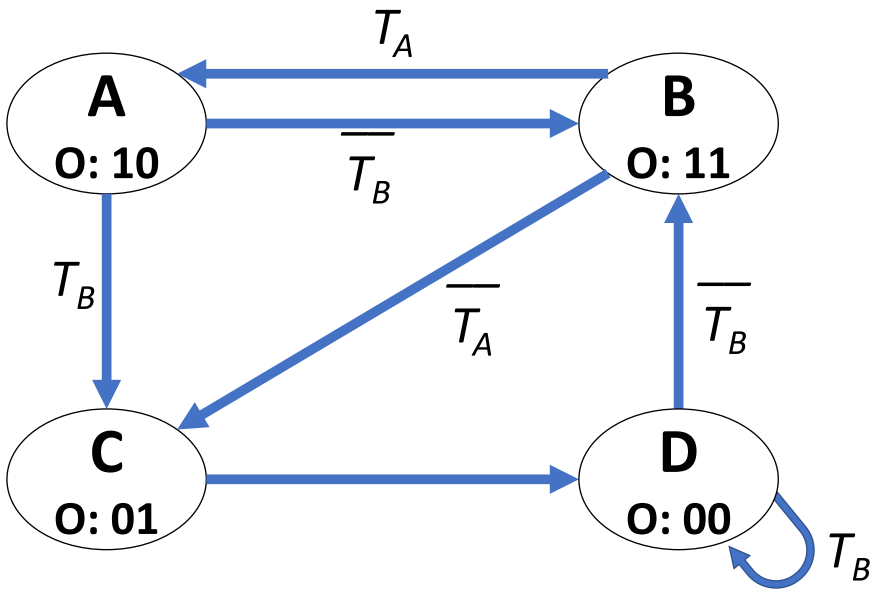
Answer and concise explanation:

Yes.

Explanation: Even though the wire *s* is swapped with the input *b*, the final computation produced by the module *adder* is still going to be correct since the *or* and *and* operations are commutative.

2 Finite State Machine

You are given the following FSM with two one-bit input signals (T_A and T_B) and one two-bit output signal (O). You need to implement this FSM, but you are unsure about how you should encode the states. Answer the following questions to get a better sense of the FSM and how the three different types of state encoding we dicussed in the lecture (i.e., one-hot, binary, output) will affect the implementation.



(a) There is one critical component of an FSM that is *missing* in this diagram. Please write what is missing in the answer box below.

The reset line or indication for initial state.

(b) What kind of an FSM is this?

Moore

(c) List one major advantage of each type of state encoding below.

One-hot encoding

reduces next-state logic

Binary (i.e., fully) encoding

reduces FFs to hold state

Output encoding

reduces the output logic

(d) Fully describe the FSM with equations given that the states are encoded with **one-hot** encoding. Assign state encodings such that numerical values of states increase monotonically for states A through D while using the **minimum** possible number of bits to represent the states with one-hot encoding. Indicate the values you assign to each state and simplify all equations:

State assignments: A: 0001, B: 0010, C: 0100, D: 1000

NS[3] = TB * TS[3] + TS[2]

NS[2] = TB * TS[0] + $\overline{TA}$ * TS[1]

NS[1] = $\overline{TB}$ * (TS[0] + TS[3])

NS[0] = TS[1] * TA

O[1] = TS[0] + TS[1]

O[0] = TS[1] + TS[2]

(e) Fully describe the FSM with equations given that the states are encoded with **binary** encoding. Assign state encodings such that numerical values of states increase monotonically for states A through D while using the **minimum** possible number of bits to represent the states with binary encoding. Indicate the values you assign to each state and simplify all equations:

State assignments: A: 00, B: 01, C: 10, D: 11

$\text{NS}[1] = \overline{TS[1]} * (\overline{TS[0]} * \text{TB} + \text{TS}[0]\ \overline{TA}) + \text{TS}[1] * (\overline{TS[0]} + \text{TS}[0] * \text{TB})$

$\text{NS}[0] = \overline{TS[1]} * \overline{TS[0]} * \overline{TB} + \text{TS}[1]$

$\text{O}[1] = \text{TS}[1]$

$\text{O}[0] = \text{TS}[1]\ \text{XOR}\ \text{TS}[0]$

(f) Fully describe the FSM with equations given that the states are encoded with **output** encoding. Use the **minimum** possible number of bits to represent the states with output encoding. Indicate the values you assign to each state and simplify all equations:

State assignments: A: 10, B: 11, C: 01, D: 00

$\text{NS}[1] = \text{TS}[1] * \overline{TS[0]} * \overline{TB} + \text{TS}[1] * \text{TS}[0] * \text{TA} + \overline{TS[1]} * \overline{TS[0]} * \overline{TB}$

$= \overline{TS[0]} * \overline{TB} + \text{TS}[1] * \text{TS}[0] * \text{TA}$

$\text{NS}[0] = \text{TS}[1] * \overline{TS[0]} + \text{TS}[1] * \text{TS}[0] * \overline{TA} + \overline{TS[1]} * \overline{TS[0]} * \overline{TB}$

$= \text{TS}[1] * (\overline{TS[0]} + \text{TS}[0] * \overline{TA}) + \overline{TS[1]} * \overline{TS[0]} * \overline{TB}$

$\text{O}[1] = \text{TS}[1]$

$\text{O}[0] = \text{TS}[0]$

(g) Assume the following conditions:

- We can only implement our FSM with 2-input AND gates, 2-input OR gates, and D flip-flops.
- 2-input AND gates and 2-input OR gates occupy the *same* area.
- D flip-flops occupy 3x the area of 2-input AND gates.

Which state-encoding do you choose to implement in order to minimize the total area of this FSM?

one-hot: 10 logic gates, 4 FFs
binary: 16 logic gates, 2 FFs
output: 10 logic gates, 2 FFs

Output encoding has the least amount of circuitry elements.

3 Big versus Little Endian Addressing

Consider the 32-bit hexadecimal number 0xcafe2b3a.

1. What is the binary representation of this number in *little endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

3a	2b	fe	ca
0	1	2	3

2. What is the binary representation of this number in *big endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

ca	fe	2b	3a
0	1	2	3

4 The MIPS ISA

4.1 Warmup: Computing a Fibonacci Number

The Fibonacci number F_n is recursively defined as

$$F(n) = F(n-1) + F(n-2),$$

where $F(1) = 1$ and $F(2) = 1$. So, $F(3) = F(2) + F(1) = 1 + 1 = 2$, and so on. Write the MIPS assembly for the `fib(n)` function, which computes the Fibonacci number $F($`n`$)$:

```
int fib(int n)
{
  int a = 0;
  int b = 1;
  int c = a + b;
  while (n > 1) {
    c = a + b;
    a = b;
    b = c;
    n--;
  }
  return c;
}
```

Remember to follow MIPS calling convention and its register usage (just for your reference, you may not need to use all of these registers):

- The argument `n` is passed in register \$4.
- The result (i.e., `c`) should be returned in \$2.
- \$8 to \$15 are caller-saved temporary registers.
- \$16 to \$23 are callee-saved temporary registers.
- \$29 is the stack pointer register.
- \$31 stores the return address.

Note: A summary of the MIPS ISA is provided at the end of this handout.

NOTE: More than one correct solution exists, this is just one potential solution.

```
fib:
addi $sp, $sp, -16 // allocate stack space
sw   $16, 0($sp)   // save r16
add  $16, $4,  $0  // r16 for arg n
sw   $17, 4($sp)   // save r17
add  $17, $0,  $0  // r17 for a, init to 0
sw   $18, 8($sp)   // save r18
addi $18, $0,  1   // r18 for b, init to 1
sw   $31, 12($sp)  // save return address
add  $2,  $17, $18 // c = a + b

branch:
slti $3,  $16, 2   // use r3 as temp
bne  $3,  $0,  done
add  $2,  $17, $18 // c = a + b
add  $17, $18, $0  // a = b
add  $18, $2,  $0  // b = c
addi $16, $16, -1  // n = n - 1
j    branch

done:
lw   $31, 12($sp)  // restore r31
lw   $18, 8($sp)   // restore r18
lw   $17, 4($sp)   // restore r17
lw   $16, 0($sp)   // restore r16
addi $sp, $sp, 16  // restore stack pointer
jr   $31           // return to caller
```

4.2 MIPS Assembly for REP MOVSB

MIPS is a simple ISA. Complex ISAs—such as Intel's x86—often use one instruction to perform the function of many instructions in a simple ISA. Here you will implement the MIPS equivalent for a single Intel x86 instruction, REP MOVSB, which is specified as follows.

The REP MOVSB instruction uses three fixed x86 registers: ECX (count), ESI (source), and EDI (destination). The "repeat" (REP) prefix on the instruction indicates that it will repeat ECX times. Each iteration, it moves one byte from memory at address ESI to memory at address EDI, and then increments both pointers by one. Thus, the instruction copies ECX bytes from address ESI to address EDI.

(a) Write the corresponding assembly code in MIPS ISA that accomplishes the same function as this instruction. You can use any general purpose register. Indicate which MIPS registers you have chosen to correspond to the x86 registers used by REP MOVSB. Try to minimize code size as much as possible.

Assume: \$1 = ECX, \$2 = ESI, \$3 = EDI

```
beq    $1, $0, AfterLoop          // If counter is zero, skip
CopyLoop:
lb     $4, 0($2)                  // Load 1 byte
sb     $4, 0($3)                  // Store 1 byte
addiu  $2, $2, 1                  // Increase source pointer by 1 byte
addiu  $3, $3, 1                  // Increase destination pointer by 1 byte
addiu  $1, $1, -1                 // Decrement counter
bne    $1, $0, CopyLoop           // If not zero, repeat
AfterLoop:
Following instructions
```

(b) What is the size of the MIPS assembly code you wrote in (a), in bytes? How does it compare to REP MOVSB in x86 (note: REP MOVSB occupies 2 bytes)?

The size of the MIPS assembly code is 4 bytes $\times$ 7 = 28 bytes, as compared to 2 bytes for x86 REP MOVSB.

(c) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0xFEE1DEAD
EDX: 0xfeed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, consider the MIPS assembly code you wrote in (a). How many total instructions will be executed by your code to accomplish the same fuction as the single REP MOVSB in x86 accomplishes for the given register state?

The count (value in ECX) is 0xfee1dead = 4276215469. Therefore, the loop body is executed 4276215469 times. As there are 6 instructions in the loop body, total instructions executed = 6 * 4276215469 + 1 = 25657292814 + 1 (beq instruction outside of the loop) = 25657292815.

(d) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0x00000000
EDX: 0xfeed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, answer the same question in (c) for the above register values.

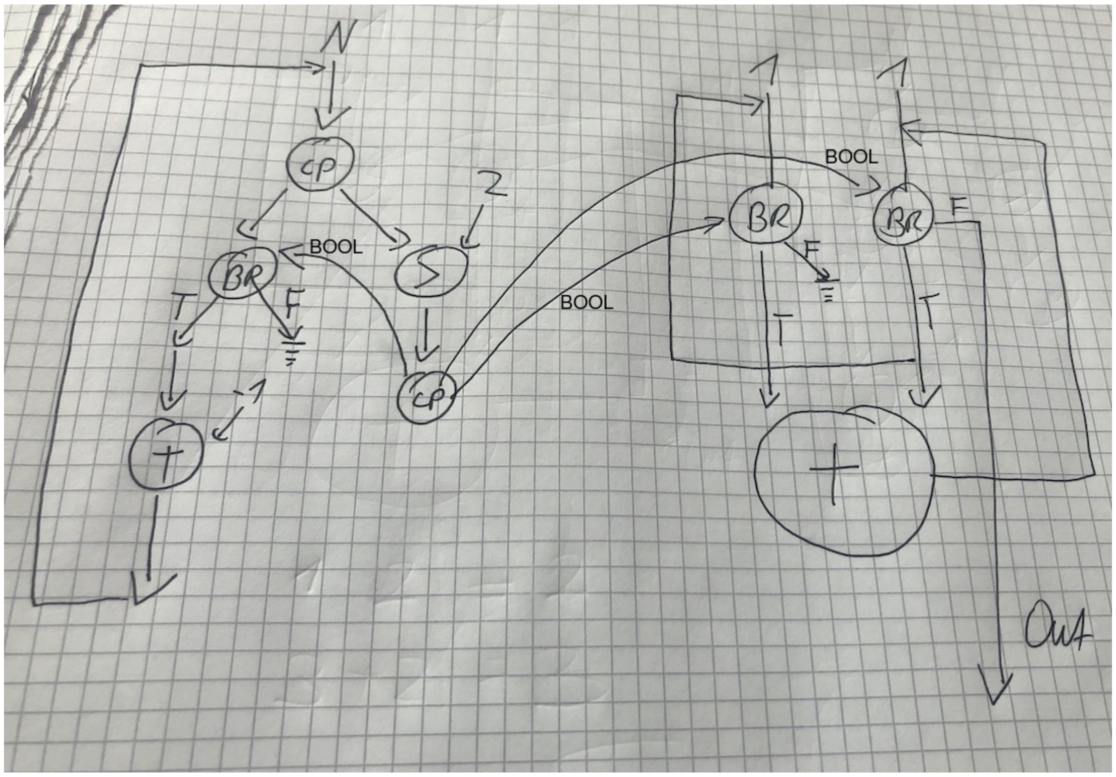
The count (value in ECX) is 0x00000000 = 0. Therefore, the loop body is executed 0 times. Total instructions executed = 1 (beq instruction outside of the loop).

5 Data Flow Programs

Draw the data flow graph for the `fib(n)` function from Question 4.1. You may use the following data flow nodes in your graph:

- + (addition)
- $>$ (left operand is greater than right operand)
- `Copy` (copy the value on the input to both outputs)
- `BR` (branch, with the semantics discussed in class, label the True and False outputs)

You can use constant inputs (e.g., 1) that feed into the nodes. Clearly label all the nodes, program inputs, and program outputs. Try to the use fewest number of data flow nodes possible.


N
CP
2
BR
BOOL
F
T
Σ
CP
+
BOOL
1
1
BOOL
BR
BR
F
F
T
T
+
Out

6 Microarchitecture vs. ISA

a) Briefly explain the difference between the *microarchitecture* level and the *ISA* level in the transformation hierarchy. What information does the compiler need to know about the microarchitecture of the machine in order to compile a given program correctly?

The ISA level is the interface a machine exposes to the software. The microarchitecture is the actual underlying implementation of the machine. Therefore, the microarchitecture and changes to the microarchitecture are transparent to the compiler/programmer (except in terms of performance), while changes to the ISA affect the compiler/programmer. The compiler does not need to know about the microarchitecture of the machine in order to compile the program correctly

b) Classify the following attributes of a machine as either a property of its microarchitecture or ISA:

Microarchitecture?	**ISA?**	**Attribute**
	✓	The machine does not have a subtract instruction
✓		The ALU of the machine does not have a subtract unit
	✓	The machine does not have condition codes
	✓	A 5-bit immediate can be specified in an ADD instruction
✓		It takes n cycles to execute an ADD instruction
	✓	There are 8 general purpose registers
✓		A 2-to-1 mux feeds one of the inputs to ALU
✓		The register file has one input port and two output ports

7 Performance Metrics

- If a given program runs on a processor with a higher frequency, does it imply that the processor always executes more instructions per second (compared to a processor with a lower frequency)? (Use less than 10 words.)

  No, the lower frequency processor might have much higher IPC (instructions per cycle).
  More detail: A processor with a lower frequency might be able to execute multiple instructions per cycle while a processor with a higher frequency might only execute one instruction per cycle.

- If a processor executes more of a given program's instructions per second, does it imply that the processor always finishes the program faster (compared to a processor that executes fewer instructions per second)? (Use less than 10 words.)

  No, because the former processor may execute many more instructions.
  More detail: The total number of instructions required to execute the full program could be different on different processors.

8 Performance Evaluation

Your job is to evaluate the potential performance of two processors, each implementing a different ISA. The evaluation is based on its performance on a particular benchmark. On the processor implementing ISA A, the best compiled code for this benchmark performs at the rate of 10 IPC. That processor has a 500 MHz clock. On the processor implementing ISA B, the best compiled code for this benchmark performs at the rate of 2 IPC. That processor has a 600 MHz clock.

- What is the performance in Millions of Instructions per Second (MIPS) of the processor implementing ISA A?

ISA A: $10\frac{instructions}{cycle} * 500,000,000\frac{cycle}{second} = 5000$ MIPS

- What is the performance in MIPS of the processor implementing ISA B?

ISA B: $2\frac{instructions}{cycle} * 600,000,000\frac{cycle}{second} = 1200$ MIPS

- Which is the higher performance processor: A B Don't know

Briefly explain your answer.

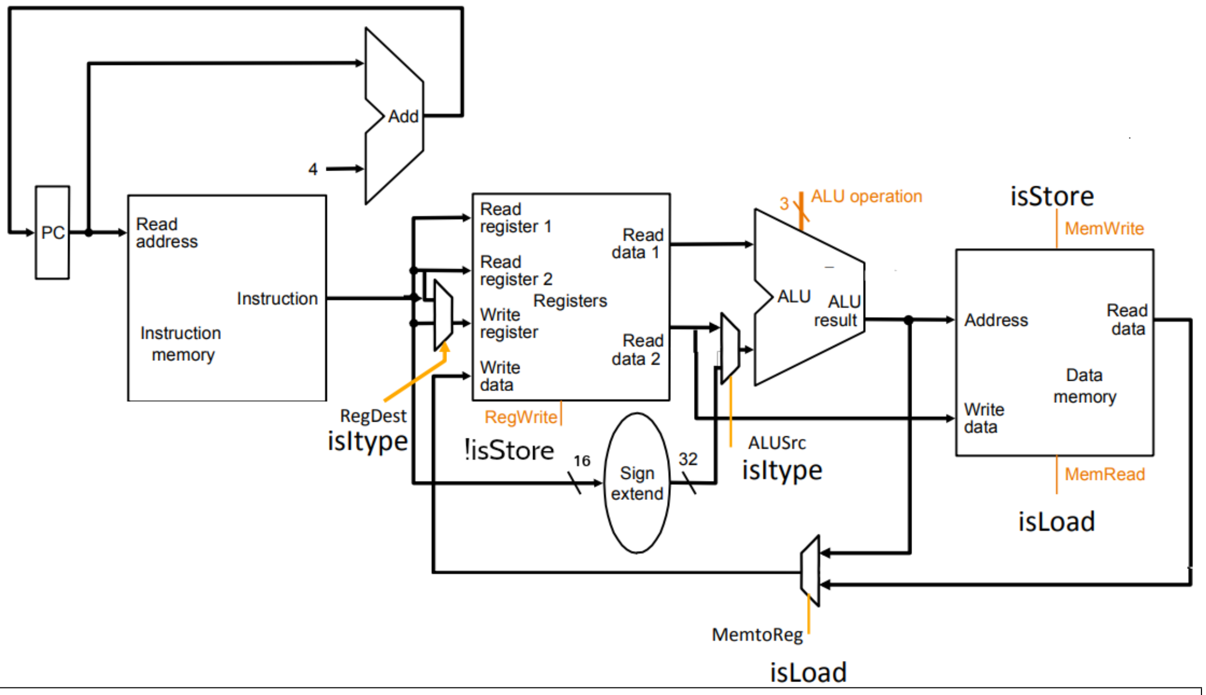
Don't know.
The best compiled code for each processor may have a different number of instructions.

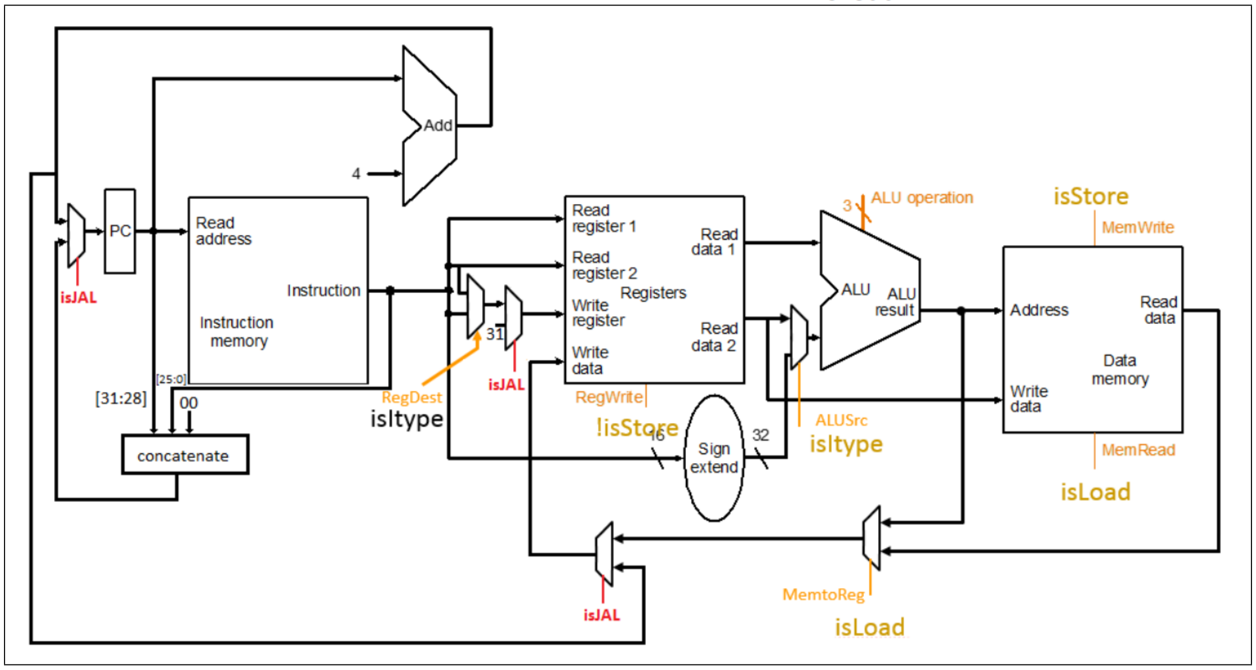
9 Single-Cycle Processor Datapath

In this problem, you will modify the single-cycle datapath we built up in Lecture 11 to support the `JAL` instruction. The datapath that we will start with is provided below. Your job is to implement the necessary data and control signals to support the `JAL` instruction, which we define to have the following semantics:

$$\begin{aligned} \texttt{JAL}: \quad & \texttt{R31} \leftarrow \texttt{PC} + 4 \\ & \texttt{PC} \leftarrow \texttt{PC}_{31...28} \,||\, \texttt{Immediate} \,||\, 0^2 \end{aligned}$$

Add to the datapath on the next page the necessary data and control signals to implement the `JAL` instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).





10 REP MOVSB

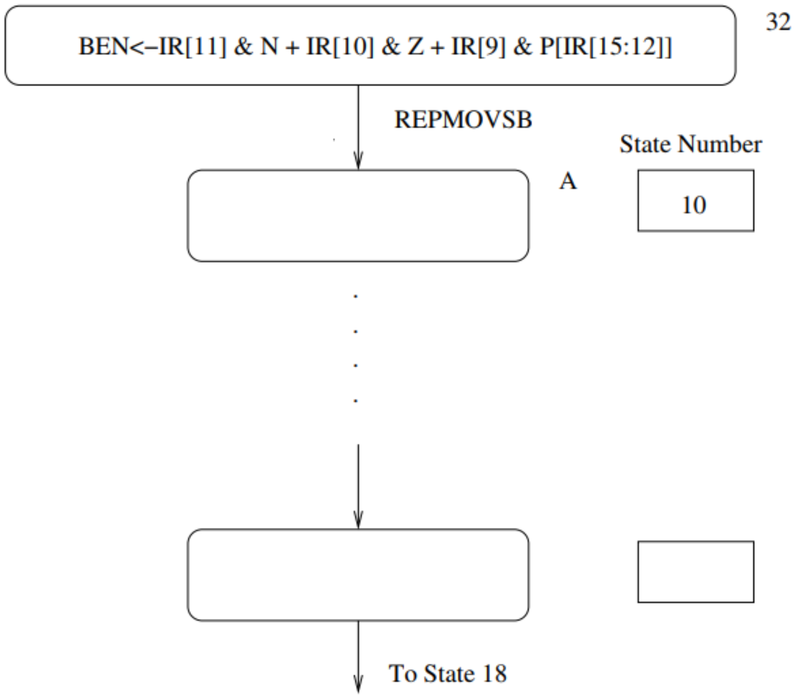
Let's say you are the lead architect of the next flagship processor at Advanced Number Devices (AND). You have decided that you want to use the LC-3b ISA for your next product, but your customers want a smaller semantic gap and marketing is on your case about it. So, you have decided to implement your favorite x86 instruction, REP MOVSB, in LC-3b.

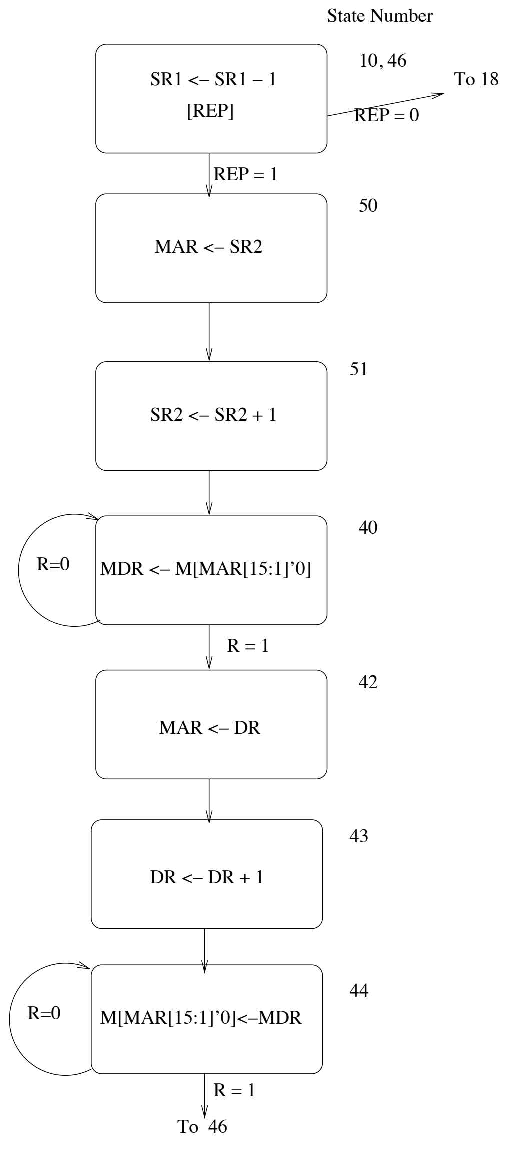
Specifically, you want to implement the following definition for REP MOVSB (in LC-3b parlance): REP-MOVSB SR1, SR2, DR which is encoded in LC-3b machine code as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010				DR			SR1			0	0	0	SR2		

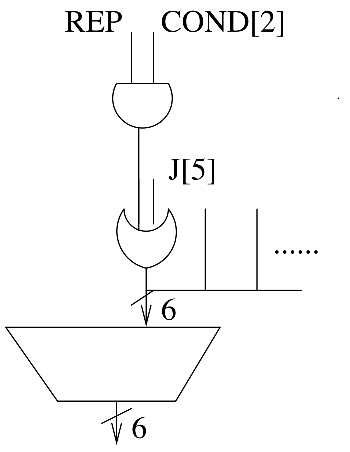
REPMOVSB uses three registers: SR1 (count), SR2 (source), and DR (destination). It moves a byte from memory at address SR2 to memory at address DR, and then increments both pointers by one. This is repeated SR1 times. Thus, the instruction copies SR1 bytes from address SR2 to address DR. Assume that the value in SR1 is greater than or equal to zero.

1. Complete the state diagram shown below, using the notation of the LC-3b state diagram. Describe inside each bubble what happens in each state and assign each state an appropriate state number. Add additional states not present in the original LC-3b design as you see fit.

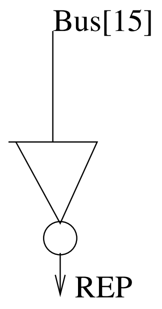


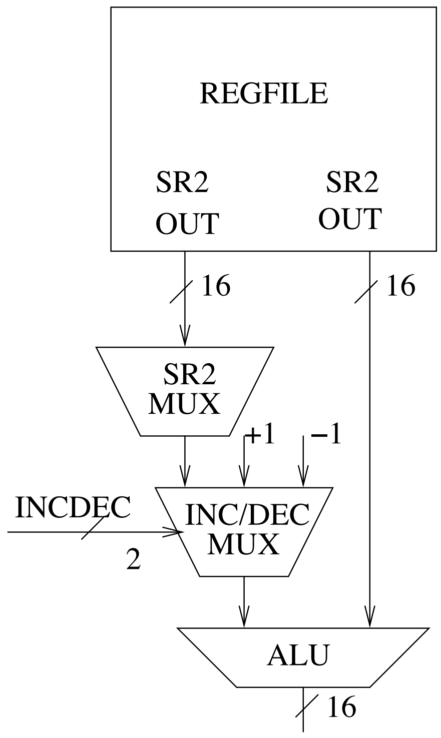

State Number
SR1 <– SR1 – 1
[REP]
10, 46
To 18
REP = 0
REP = 1
MAR <– SR2
50
SR2 <– SR2 + 1
51
R=0
MDR <– M[MAR[15:1]'0]
40
R = 1
MAR <– DR
42
DR <– DR + 1
43
R=0
M[MAR[15:1]'0]<–MDR
44
R = 1
To 46

2. Add to the LC-3b datapath any additional structures and any additional control signals needed to implement REPMOVSB. Clearly label your additional control signals with descriptive names. Describe what value each control signal would take to control the datapath in a particular way.

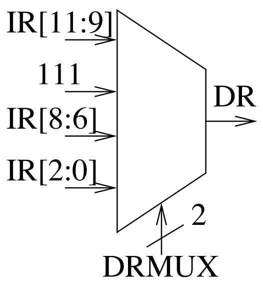


Microsequencer Modifications

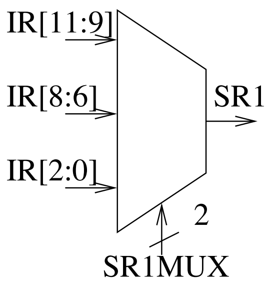




Regfile Modifications



DRMUX Modifications



SR1MUX Modifications

3. Describe any changes you need to make to the LC-3b microsequencer. Add any additional logic and control signals you need. Clearly describe the purpose and function of each signal and the values it would take to control the microsequencer in a particular way.

Additional control signals

- INCDEC/2: PASSSR2, +1, -1
- DRMUX/2:
  - IR[11:9] ;destination IR[11:9]
  - R7 ;destination R7
  - IR[8:6] ;destination IR[8:6]
  - IR[2:0] ;destination IR[2:0]
- SR1MUX/2:
  - IR[11:9] ;source IR[11:9]
  - IR[8:6] ;source IR[8:6]
  - IR[2:0] ;source IR[2:0]
- COND/3:
  - COND0: Unconditional
  - COND1: Memory Ready
  - COND2: Branch
  - COND3: Addressing Mode
  - COND4: Repeat

MIPS Instruction Summary

Opcode	Example Assembly	Semantics
add	`add $1, $2, $3`	`$1 = $2 + $3`
sub	`sub $1, $2, $3`	`$1 = $2 - $3`
add immediate	`addi $1, $2, 100`	`$1 = $2 + 100`
add unsigned	`addu $1, $2, $3`	`$1 = $2 + $3`
subtract unsigned	`subu $1, $2, $3`	`$1 = $2 - $3`
add immediate unsigned	`addiu $1, $2, 100`	`$1 = $2 + 100`
multiply	`mult $2, $3`	`hi, lo = $2 * $3`
multiply unsigned	`multu $2, $3`	`hi, lo = $2 * $3`
divide	`div $2, $3`	`lo = $2/$3, hi = $2 mod $3`
divide unsigned	`divu $2, $3`	`lo = $2/$3, hi = $2 mod $3`
move from hi	`mfhi $1`	`$1 = hi`
move from low	`mflo $1`	`$1 = lo`
and	`and $1, $2, $3`	`$1 = $2 & $3`
or	`or $1, $2, $3`	`$1 = $2 | $3`
and immediate	`andi $1, $2, 100`	`$1 = $2 & 100`
or immediate	`ori $1, $2, 100`	`$1 = $2 | 100`
shift left logical	`sll $1, $2, 10`	`$1 = $2 << 10`
shift right logical	`srl $1, $2, 10`	`$1 = $2 >> 10`
load word	`lw $1, 100($2)`	`$1 = memory[$2 + 100]`
store word	`sw $1, 100($2)`	`memory[$2 + 100] = $1`
load upper immediate	`lui $1, 100`	`$1 = 100 << 16`
branch on equal	`beq $1, $2, label`	`if ($1 == $2) goto label`
branch on not equal	`bne $1, $2, label`	`if ($1 != $2) goto label`
set on less than	`slt $1, $2, $3`	`if ($2 < $3) $1 = 1 else $1 = 0`
set on less than immediate	`slti $1, $2, 100`	`if ($2 < 100) $1 = 1 else $1 = 0`
set on less than unsigned	`sltu $1, $2, $3`	`if ($2 < $3) $1 = 1 else $1 = 0`
set on less than immediate	`sltui $1, $2, 100`	`if ($2 < 100) $1 = 1 else $1 = 0`
jump	`j label`	`goto label`
jump register	`jr $31`	`goto $31`
jump and link	`jal label`	`$31 = PC + 4; goto label`