COMP22111

Two hours

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Processor Microarchitecture

Date: Tuesday 19th January 2016

Time: 14:00 - 16:00

Please answer any THREE Questions from the FOUR Questions provided

Use a SEPARATE answerbook for EACH Question.

This is a CLOSED book examination

The use of electronic calculators is permitted provided they are not programmable and do not store text

[PTO]

1.

a) The Stump processor has a load/store architecture. Briefly explain what this means and how the Stump processor supports this type of architecture. (2 marks)

b) The Stump processor has three instruction formats. Briefly discuss the difference between Type 1 and Type 2 instructions. Provide example instructions for Type 1 and Type 2. (4 marks)

How does the Stump control differentiate between Type 1 and Type 2 instructions? (2 marks)

What operation do Type 3 instructions perform? Explain the format of a Type 3 instruction, detailing the different bit-fields in the instruction. (3 marks)

c) Stump has a limited instruction set; however, it may be possible to implement additional operations. Explain how the following could be implemented by the Stump assembler. Give an example instruction in each case.

i) transfer of the contents of one register to another register (MOV instruction), (2 marks)

ii) a left shift of the contents of a register by 1 bit. (2 marks)

d) Produce some Stump assembly code that adds two 32-bit numbers. The two 32-bit values to be added are stored in consecutive locations in memory and are indicated as *Value1* and *Value2* in the memory map of Figure 1. The label **myData** illustrated in Figure 1, points to the least significant 16-bits of the first value, *Value1*. The result produced by adding the two values must be stored to the memory location referenced by the label **myResult** in Figure 1, with the least significant 16-bits stored first. A list of the Stump mnemonic formats can be found at the end of the examination paper. (5 marks)

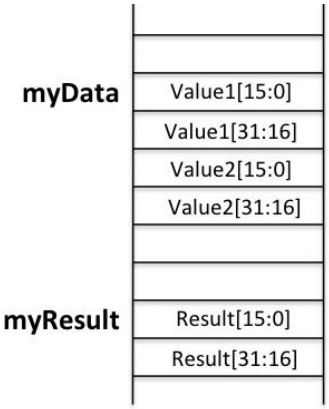

Figure 1: Location of variables in memory

2.

a) What type of circuits do you associate the use of

i) blocking statements, and (1 mark)
ii) non-blocking statements (1 mark)

in Verilog code?

b) Using a suitable diagram discuss how the simulator will execute an `always` block containing a list of non-blocking statements. (4 marks)

c) The Verilog code given in Figure 2 represents a FSM used in the design of a simple RISC processor.

```
module Stump_FSM(  input              clock,
                   input              reset,
                   output  reg  [1:0] state);

always @ (posedge clock)
  if (reset == 1)  state <= 0;
  else
    case (state)
      2`b00:       state <= 2`b01;
      2`b01:       state <= 2`b10;
      2`b10:       state <= 2`b00;
      default:     state <= 2`hx;
    endcase

endmodule
```

Figure 2: RISC processor FSM

Explain the purpose of:

i) the `reg` declaration (1 mark)
ii) the `always` block (1 mark)
iii) the `default` statement (1 mark)

Translate this FSM design into a state transition diagram. (2 marks)

d) Figure 3 illustrates a simple block diagram of the datapath and control for a design of an unsigned multiplier that takes two 8-bit numbers, A and B, and produces a 16-bit result, S.

(Question 2 continues on the following page)

(Question 2 continues from the previous page)

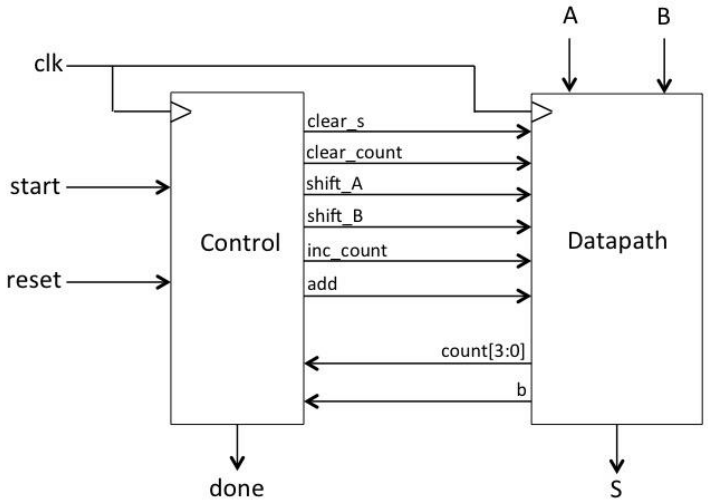

Figure 3: Multiplier datapath and control

The state transition diagram in Figure 4 depicts the behaviour of the FSM in control block of the multiplier. Produce a Verilog module that will implement the required control behaviour for the multiplier. Note: reset is asynchronous and should reset the FSM to the initial state, 0, for a 0 to 1 transition on the input reset.

(9 marks)

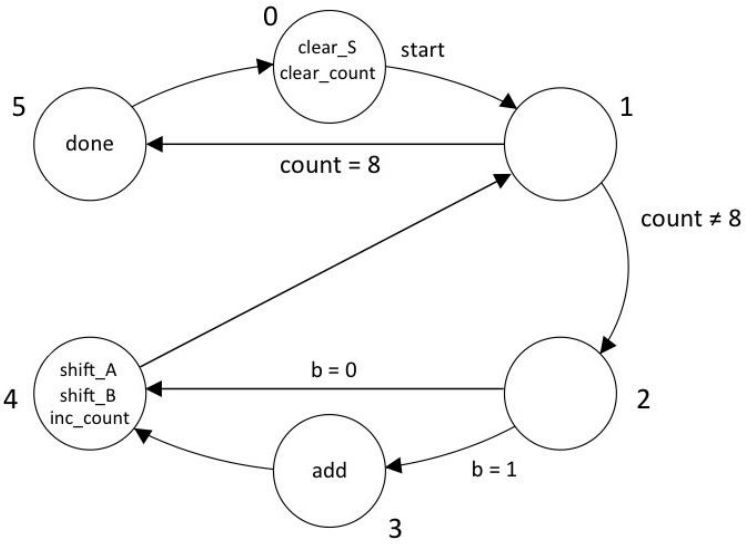

Figure 4: State transition diagram for the binary multiplier

3.

A RISC processor with a load/store architecture (like Stump, ARM etc.) contains the following (indivisible) blocks which are used successively in the execution of an instruction. Their critical path timings are also given.

Fetch	6 ns
Decode	4 ns
Register read	3 ns
Execute	5 ns
Data memory access	6 ns
Register write	2 ns

a) Assuming a single memory bus, what is the *minimum* number of cycles for:

i) an internal, register based ALU operation?
ii) loading a register from memory?

In each case, state what the minimum cycle time *could* be. Treat the operations independently in this part of the question, i.e. the cycle times *may* be different. (6 marks)

b) Use your answer to part (a) to determine the faster implementation using a constant clock period, given that 20% of the operations are loads/stores, on average. (6 marks)

c) Explain how **pipelining** can, under some circumstances, accelerate the performance of a processing system. (2 marks)

d) Partition the processing blocks above into the best possible 4-stage pipeline. The constraint on memory buses can be removed for this purpose, however each pipeline register adds a 1 ns delay. What is the minimum clock period? (6 marks)

[PTO]

4.

a) Explain how a FET (Field Effect Transistor) operates as a switch in a MOS (Metal-Oxide-Semiconductor) digital circuit. (2 marks)

b) What different types of MOS FETs are used in CMOS circuits? Show, with the aid of a schematic diagram, how these FETs are used to form a two-input NAND gate. (4 marks)

c) Why are all single-stage CMOS gates inverting (i.e. NOT, NAND, NOR)? (2 marks)

d) What is meant by the 'fan out' and the 'fan in' of a CMOS gate? (4 marks)

e) When a CMOS gate switches, the output state will not change instantaneously. Why is it important to establish this output edge speed -after- the gates have been placed and routed in the chip layout? (4 marks)

f) If the edge speed determined by part (e) is deemed too slow, suggest what might be done to fix the problem. (4 marks)

END OF EXAMINATION

Stump Instruction Formats

Data operations

```
ADD{S}     <Rdest>, <RsrcA>, <Rsrcb> {, shift)
ADC{S}     <Rdest>, <RsrcA>, <Rsrcb> {, shift)
SUB{S}     <Rdest>, <RsrcA>, <Rsrcb> {, shift)
SBC{S}     <Rdest>, <RsrcA>, <Rsrcb> {, shift)
AND{S}     <Rdest>, <RsrcA>, <Rsrcb> {, shift)
OR{S}      <Rdest>, <RsrcA>, <Rsrcb> {, shift)
ADD{S}     <Rdest>, <RsrcA>, #<expr>
ADC{S}     <Rdest>, <RsrcA>, #<expr>
SUB{S}     <Rdest>, <RsrcA>, #<expr>
SBC{S}     <Rdest>, <RsrcA>, #<expr>
AND{S}     <Rdest>, <RsrcA>, #<expr>
OR{S}      <Rdest>, <RsrcA>, #<expr>
```

Memory transfers

```
LD    <Rdest>, [<RsrcA>]
ST    <Rdest>, [<RsrcA>]
LD    <Rdest>, [<RsrcA>, #<expr>]
ST    <Rdest>, [<RsrcA>, #<expr>]
LD    <Rdest>, [<RsrcA>, <RsrcB>]
ST    <Rdest>, [<RsrcA>, <RsrcB>]
LD    <Rdest>, [<RsrcA>, <RsrcB>, shift]
ST    <Rdest>, [<RsrcA>, <RsrcB>, shift]
LD    <Rdest>, [PC, label]
ST    <Rdest>, [PC, label]
LD    <Rdest>, label
ST    <Rdest>, label
```

Control transfer

```
bal   label      ; Always
b     label      ; Always (alternative)
bra   label      ; Always (alternative)
bnv   label      ; Never (uninteresting)
bhi   label      ; HIgher
bls   label      ; Lower or Same
bcc   label      ; Carry Clear
bcs   label      ; Carry Set
bne   label      ; Not Equal
beq   label      ; EQual
bvc   label      ; oVerflow Clear
bvs   label      ; oVerflow Set
bpl   label      ; PLus (positive)
bmi   label      ; MInus (negative)
bge   label      ; Greater or Equal
blt   label      ; Less Than
bgt   label      ; Greater Than
ble   label      ; Less or Equal
```