Please see the attached.
COMP22111 Exam Feedback 2014/15

P W Nutter, & J D Garside

The average mark for the paper was 50%, with almost all students answering questions 1 and 2 – the average for these questions were 61% and 56% respectively. However, questions 3 and 4 were poorly answered overall, with averages of 28% and 39% respectively, which brought the average of the overall exam down. The average for the lab was 63%, which will bring the overall average mark up slightly.

General comments. Please write your answers neatly, as marking is very difficult if we can’t read what you have written, or if we can’t clearly identify your answer! You can have as many answer booklets as you like, so please don’t squeeze your answer into the smallest space, on the corner of a page!

Q1 (PWN)

This question was largely bookwork, apart from part e) where you have done something similar in labs. Hence, I thought this would be relatively well answered, which it was.

a)

There was an error in the question, as some of you pointed out. It should have said CISC in the brackets, not RISC).

I expected a brief discussion of two distinguishing features of RISC processors, such as:

- fixed instruction size, with a limited number of well defined instructions,
- load/store architecture where the processor operates on data held in registers and writes the result back to a register
- greater on chip memory via a local register bank where data operands are located
- the processor cannot operate on data held in memory, load and store instructions are provided to access data in memory

1 mark was awarded for each (sensible) answer. There were no real problems in answering this question.

b)

This question required you to describe two addressing modes used by Stump, giving example instructions for each, for example:

- immediate – instruction contains a literal value as one of the operands, example: ADD R2, R3, #4
• register – operands are held in registers specified in the instruction, example: SUB R5, R4, R3
• offset – location of an operand in memory is specified by a register (identified in the instruction) and a literal offset value given in the instruction, which are added to give the memory address, example: LD R4, [R1, #1]
• PC relative – instruction contains a literal value that is an offset to the PC, used in branch instruction, example: BAL #5
• Indexed – the instruction specifies two registers containing operands that are added together to give the address of the location of data in memory, example: ST R3, [R6, R2].

Some problems/misconceptions:

• Not giving Stump code examples, as requested.
• Mixing up cases where the data is an operand, or the data is an address.
• Not stating how the register or immediate value is specified, i.e. in the instruction!
• Mixing up examples of Stump code for the different types od instruction. For example, giving an example of indexed addressing for register addressing. Both cases refer to a register, however, the contents of the registers are different for the two cases (see previous comment). A register addressing example would not be LD/ST as these are dealing with addresses, so are indexed, or offset examples.

c)

Easy question that on the whole was answered well. The two special registers are: R0, which is hardwired to 0 to allow instructions such as MOV and NOP to be implemented, and R7, which is the program counter used to keep track of the address of the next instruction in memory. 1 mark for each, ½ a mark for identifying the special register, ½ a mark for stating why it is special.

Some problems/misconceptions:

• Not identifying why R0 is wired to 0.
• Not stating that the PC is R7.
• Incorrectly stating that the PC holds the address of the current instruction. It does in fact hold the address to the next instruction.

d)

On the whole this question was answered well. The first part of the question asked you to identify what the N, Z, V, C registers are used for.
• N – set to 1 if the result of an operation is negative.
• Z – set to 1 if the result of an operation is zero.
• V – set to 1 if the result of an operation is invalid, or leads to an overflow.
• C – set to 1 if the result of an operation results in a carry out of bit 15).
The second part of the question asked you to identify how the N, Z, and V flags
are determined knowing the operation, the operands and the result.

- Z is determined by checking the result is zero, or not. This can be done by ORing all the bits.
- N is determined by looking at the state of the most significant bit, bit 15, of the result.
- V can be determined by comparing the sign of the operands, and the sign of the result, to determine if the result is the correct sign depending on the operation performed. I also accepted an answer that discussed comparison of the carry outs from bit 14 and bit 15 of the result.

Note: the question did not ask for you how the carry would be determined. Each correct answer was worth ½ mark, apart from how you would determine V, which was awarded 1 mark.

Some problems/misconceptions:

- Not stating how the flags are generated for N, Z, and V.
- Discussing how C is generated – you were not asked to do this!
- For the N flag, stating that the flag is set depending on whether the result is negative or not. How do you determine if it is negative? This is what the question was asking!
- The Z flag can be generated by XORing, or ANDing all the bits in the result – no it won’t! Think about it.

e)

This question asked you to identify which registers changed, and what the new values were, after each instruction in the code snippet is executed. You were required to take into account the instructions in the order they would be executed. There were some worrying answers given here that made it clear that some students did not understand how the Stump operates, even though its design is covered extensively in lectures and the lab!

Where a mistake was made in evaluating an instruction, I did not penalize in later instruction where the error was propagated.

Some problems/misconceptions:

- One major omission was the fact that the PC changes for EVERY instruction, a large number of answers failed to recognize this fact and consequently up to 4 marks were lost!
- Treating ORG 0x101 as an instruction. It isn’t an instruction, it is a precompiler directive to tell the compiler where the following code should be located in memory, i.e. starting at address 101.
- The address given by the ORG statement is in hex, not binary, so there were some cases where the PC was given where the address was incremented in binary – I wasn’t too hard on this.
- For the instruction ADD R0, R1, #0x4 a number of answers stated that R0 is updated, even though in answer to the earlier question is was stated that R0 is hard-wired to zero!
For the instruction ADCS R5, R3, R5 is was not recognized that the carry flag is set, so 1 needs to be added along with the contents of R3 and R5, so in some cases the answer was out by one!

- Not recognizing that the result of the instruction ADC R5, R3, R5 is negative, which result in the N flag being set (due to the ‘S’ in the instruction). This happened in other cases, so there was clearly some misunderstanding what the ‘S’ means!

- Setting individual flags, instead of re-evaluating all flags. For example in the case of ADCS R5, R3, R5, only the negative flag should be set, all other flags should be cleared. However, in a number of cases, only the N flag was updated, and the C flag remained set (as it was initially). However, this could have been left set as there was some belief that the carry flag should be set for the following branch instruction, who knows! However, this isn’t the case, as the branch should not be taken.

- Not taking into account in later instructions that the contents of a register may have been updated in a previous instruction, such as R5 in the instruction SUBS R4, R5, R1, which was updated in the earlier instruction ADCS R5, R3, R5. Again, this was a devious trap I set that some of you fell into! (Maniacal laugh!)

- None of the branches are taken, so all the instructions will be executed.
- Not clearly stating that a branch is not taken, or that the PC is still updated by 1 for a branch instruction if not taken.

Q2 (PWN)

This question tested your understanding of the Stump; something that you should know very well from the labs and early lectures. In some cases I was shocked by the level of understanding!

a)

This question asked you to discuss, using the key blocks in the datapath, the operation of each of the types of instruction. On the whole this was answered well.

Some problems/misconceptions:

- Going into detail about the fetch process – this wasn’t required.
- Not clearly stating where the operands come from for each instruction, i.e. when operands/addresses are stored in registers, then a register is identified by number in the instruction.
- Not mentioning the operation of the shifter for the Type 1 instruction.
- Not stating that the result is written back to a register in the register bank for the case of Type 1 and Type 2 instruction (but not LD/ST instructions).
- Not being clear that the shifter does not do anything in the case of the Type 2 instruction (some even stated that a shift was applied for a Type 2 instruction!).
- Treating the literal value from the instruction in the case of a Type 3 instruction as the address to be written to the PC if the branch is taken. The
literal value is in fact an offset from the PC, so the PC must be passed from the register bank as operand A through to the ALU unshifted.

- Not being clear what happens in the case where a branch is not taken.
- Stating that the flags are evaluated and depending on the result the branch instruction is executed or not. A Type 3 instruction is ALWAYS executed, the control logic determines whether the PC should be enabled for the writeback process.
- Not identifying that LD/ST instructions can be of Type 1 or Type 2.
- Not clearly stating what happens during the LD/ST memory phase.

b)

An easy question, but some shocking answers! The question asked you to describe the operation of the shifter and sign extender blocks in the case of the Stump (as it refers to the Stump datapath given in figure 2). Sensible answers are:

- The shifter applies one of 3 shift operation, or not, depending upon the status of bits [1:0] of the instruction in the case of a Type 1 instruction (1 mark).
- The sign extender sign extends from 5 bits (in the case of a Type 2 instruction), or 8 bits (in the case of a Type 3 instruction) to 16 bits so that a 16-bit operation can be performed by the ALU. The sign extension is performed by copying the sign bit of the 5 bit, or 8 bit value over all the higher bits – 1 mark.

Each answer was worth 1 mark. I was generous in my marking.

Some problems/misconceptions:

- Stating that the shifter does right or left shifts. In the case of the Stump example shown in figure 2, which the questions refers to, the shift operation is only 1-bit to the right. However, how bit 15 in the shifted output, and the value of CSH differ between the 3 different shift operations.
- Not clearly stating the different types of shift that can be applied, or even simply stating there were three different types!
- Not, or incorrectly, stating the number of bits (i.e. 5 or 8) being sign extended.

c)

For this question you were required to write some Verilog code. Something you have had a lot of experience of in the lab. Some of the answers shocked me, and in some cases there is clearly an inability to write sensible, coherent, syntactically correct Verilog code! The sign extender module was worth 5 marks, 3 marks for functionality (i.e. functionally it does what expected) and 3 marks for Verilog style/syntax. The shifter was worth 9 marks, 6 for functionality, 3 for Verilog style/syntax.

There was a mistake in the codes for the shifter given in figure 3, so I didn’t mark anyone down if these errors were translated into the Verilog implementation.
Example code:

**Sign extender:**

```verilog
module sign_extender(input  [15:0] IR,
                      output reg [15:0] immediate);

always(*)
  if(IR[15:13] == 3'b111) // then we have 8-> 16 bit
    immediate[15:0] = {8{IR[7]}}, IR[7:0]);
  else
    immediate[15:0] = {{11{IR[4]}}, IR[4:0]);
endmodule
```

**Shifter:**

```verilog
module shifter( input  [15:0] data_in,
                input  [1:0] shift_op,
                input CC0,
                output reg [15:0] data_out,
                output reg CSH);

always@(*)
  case(shift_op)
    00: begin
      data_out = data_in; // no shift
      CSH = 0;
    end
    01: begin
      data_out = data_in >> 1; //ASR
      data_out[15] = data_in[15];
      CSH = data_in[0];
    end
    10: begin
      data_out = data_in >> 1; //ASR
      data_out[15] = data_in[0];
      CSH = data_in[0];
    end
    11: begin
      data_out = data_in >> 1; //ASR
      data_out[15] = C[0];
      CSH = data_in[0];
    end
    default: begin
      data_out = 16’hxxxx // trap errors
      CSH = 1’bx;
    end
  endcase
```
Some problems/misconceptions:

- Not including an “always” block in the module. Unless you are performed continuous assignments (using assign), then all assignments must be performed in an always block – basic rules of Verilog.

- Defining the module incorrectly, and/or defining inputs as reg and not defining outputs as reg. Remember, inputs are not defined, they are wires, which are the default type, and hence do not need defining explicitly. Outputs, if assigned using blocking, ‘=’, statements, need to be defined as reg.

- Missing the ; from the end of the module definition (outside of the brackets).

- Putting posedge clock, or reset, in the sensitivity list of the always block (where is clock or reset in the interface to the module?).

- Using the wrong type of assignments. The blocks are combinatorial, so blocking statements, i.e. ‘=’, are used. If posedge clock was wrongly used in the sensitivity list, then I wasn’t negative in my marking in non-blocking assignments were used.

- Using inputs and outputs that do not exist in the datapath design given. The design of the Stump datapath given in figure 2 is deliberately different from that used in the lab, to make you think! There is no control signal for the sign extender that tells it how to sign extend. This can in fact be derived directly from the instruction itself by looking at bit 15:13 (which will be 111 for a Type 3 instruction), which is what the questions hints at, and why all 15 bits of the instruction are passed to the sign extender in this design.

- Using ‘for’ statements in the module; these cannot be synthesized into hardware, and as such, are invalid!

- Not recognizing that it is simple to do a shift in Verilog. The code:

  ```
  data_out = data_in >> 1;
  ```

  shifts the contents of data_in one bit to the right into data_out, data_in remains unchanged.

- Not setting both data_out and csh for all cases, including the no shift case.

- Making up random Verilog keywords, examples being beginmodule, endalways, begincase. My heart sank!

- Clearly not knowing how sign extension works. Copying just the 5 bit literal or the 8 bit literal into the output does not implement a sign extender!

- Not knowing how to do concatenation in Verilog, as this makes sign extension much easier, for example:

  ```
  immediate[15:0] = {{8{IR[7]}}, IR[7:0]};
  ```

  copies the sign bit of the immediate (bit[7]) over all 8 MSBs of the output. I wasn’t too unduly negative in my marking of this. Where this was attempted, I tended to give all the marks available.

- Missing the begin and end when multiple assignments are performed, particularly for the different cases in case statements where used.
- Not setting CSH for all cases.
- General syntax errors – missing endcase, endmodule etc.
- Incorrectly putting begin and end inside a case statement, i.e.

  ```verilog
case(shift_op)
  begin
    2'b00:
    ...
  end

  There should be no begin and end here, this is invalid Verilog.
```
Q3 (JDG)

Marking this question turned out to be thoroughly depressing. 19 candidates submitted answer books (one was completely blank, some more almost so) and the average mark was below 5.4/20. I still don’t believe the question to be particularly hard in the main and only the final ~20% of the marks require deep interpretation.

a) Simple 'bookwork' which quite a number of people knew.

b) A followup to the first part with a bit more knowledge required but some choice of an answer. Some answers about one or other (usually von Neumann) being "simpler" - which it isn't because there is no need to multiplex the instruction fetches and the data stream. Several answers stating one or other is easier to pipeline: again why would that be? Pipelining is largely orthogonal to the memory architecture.

c) I believe any CS student should comprehend that 2^16 = 64K. Judging from the evidence here 11/19 ~60% of hardware specialists have grasped this.

d) At this stage of the question some deduction is needed. The question part is about the instruction set encoding, something widely seen (on a different ISA) in the lab. There was a hint inserted before the particular question was asked, giving all the necessary information (i.e. no specific extra details required). There were many references to speed - something which could only be justified by reducing the number of cycles and the question states the instruction size is (effectively) fixed. The short addresses are there because two or more 16-bit addresses won't pack into a 16-bit instruction.

e) Those who had got this far usually made a reasonable - if sometimes confused - answer here.

f) The main part of the question which is partly interpolation and partly assembling knowledge from other parts of the module. The relatively few answers submitted were mostly confused. The desired figure is an interpretation of one in the course notes but all the factual information needed was in the question. The cross-overs into bus structure - registers want parallel reads & write – and implementation technology were largely lost.

Q4 (JDG)

Treated largely as an alternative to Q3. Not great but better answered with an overall average ~7.9/20. Candidates clearly tend to avoid the physical design questions: it can be observed that many also neglected the lectures. With some sections of this question - in particular part c) but in other places too - there was a tendency to spot a key word/phrase and write about that, rather than addressing the question itself. This question was intended to bridge several topics so although it focuses on physical layout it should draw in aspects of tool
flow, economics, Moore's Law etc.

a) This is not intended as a 'trick' (as one candidate believed) but as a simple problem to see if people appreciate what is meant when the technology is discussed. Devices are placed in 2D on the chip so the number scales with the square of the linear dimensions. The majority seemed -not- to appreciate this.

b) A bit vague in places but generally not too badly answered. There are more than three possible answers. A number of people seemed to think that 'lower power' and 'less heat dissipated' are somehow distinct points. The most surprising thing was that few people plumped for the 'smaller chip is cheaper' answer, which is a simple point.

c) Most people seemed to appreciate what standard cells are - but that wasn’t the question asked! They are present for convenience, saving designer time and improving portability. The do not give best area efficiency (else why would anyone use anything else?); they do tessellate but that does not mean custom logic cannot - it depends what the designer does.

d) A bit 'bluffed' in many cases. A disappointing number of answers including the keyword "density".

e) This part of the question was generally not interpreted as it was intended, which was to look at the -physical- parameters extracted. Most interpreted this as asking for the feedback to the designer. It was marked on that basis as appropriate as it may be seen to be ambiguous. On that basis, most answers were okay although -timing- was not usually the emphasised characteristic.

f) Reasonably well answered.

g) This part of the question was intended to be difficult and require some significant reasoning. There were a lot of attempts which produced numerous (conflicting?) answers. There were some - more than might have been anticipated – valid attempts to reason out an answer and this is laudable. Unfortunately most were based on flawed appreciation of the physical characteristics chips et al. Thus, many thought the wires become longer whereas by packing into 3D the average distances between components should reduce (else what would be the point?). Thus, sadly, the confusion about what could happen meant that the typical mark was not high. The question asks about - power. Reducing wire length will reduce the capacitance (thus the energy requirement). However the power density may rise due to the volume reduction and the surface area will fall thus, probably raising the temperature.