yz709-CD-sup1

Question 1 (y2010-p5-q2)
Question 2 (y2017-p5-q1)
Question 3 (y2016-p5-q1)
Question 4
Question 5
Question 6
Question 7 (y2021-p5-q1)
Question 8
Question 9
Question 10
Question 11

Question 1 (y2010-p5-q2)

Gordon Moore's "law" was originally an observation about transistor density improving exponentially and the implications for the semiconductor industry.

- (a) Does Moore's law still apply to transistor density? Justify your answer. [4]
  - Moore's law indicates the transistor density would double every two years, which is still valid.
  - However, the physical scaling of gates on the transistor has a limit because as the transistor shrinks, the control of the current flow becomes difficult.
  - For a transistor taken the form of a MOSFET, further physical scaling would cause source-to-drain leakage.
- (b) Can Moore's law be applied to processor performance? Justify your answer. [4]
  - Suppose we combine Moore's law with Dennard scaling, which indicates that when the transistor gets smaller, the power consumption density stays the same and halves as area halves. In that case, we can imply that with a double number of transistors on chips with the same area, the performance would double while power consumption keeps constant.
  - However, the Dennard scaling broke down around 2006. Because a smaller transistor poses a current leakage challenge and causes the chip to heat up, it increases static power consumption.
  - Moore's law can be applied to performance before 2006 but not after 2006.
- (a*) Describe Moore's law. Is it still valid today?

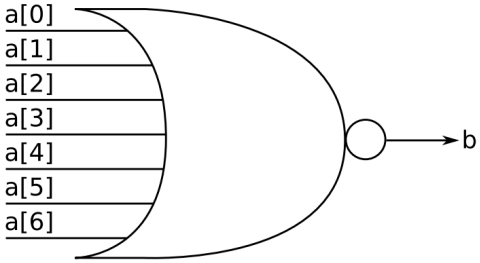
    - Moore's law indicates that the transistor density would double every two years, though the cost of computers is halved.
    - It is still valid that the transistor density would double every two years, but there might be physical scaling limits in the future. However, the consistent power consumption no longer holds.
- (b*) Explain the relationship between Moore's law and the performance of processors. Explain the relationship between Moore's law and Dennard's scaling before 2006 and after 2006.
    - Combine Moore's law with Dennard's scaling, a double number of transistors with constant power consumption density implies that the performance would double every two years.
    - Dennard's scaling:
        - Before 2006, in every technology generation where the transistor density doubles, the circuit becomes 40% faster, and the power consumption with twice the number of transistors stays the same. So it aligns with Moore's law.
        - After 2006, semiconductor technology no longer delivers on Moore's law for performance because a smaller transistor poses a current leakage challenge and causes the chip to heat up, increases static power consumption.
- (c*) What are the real-world consequences of Dennard's scaling breakdown?
    - CPU manufacturers focus more on processor architecture (e.g., multicore processor) to improve performance.
    - Constrain the number of switching transistors, domain-optimised accelerator turns on only when needed, different cores for different workloads, the remaining inactive area on a chip is called dark silicon.
    - Use power-saving strategies such as Race-to-dark that cores run at their highest operating frequency to enter the lowest operating frequency state as soon as possible, saves static and dynamic power.
- (d*) Explain the phenomena of dark silicon.
    - The transistor density would still double every two years, but the power consumption with double the transistors would not be consistent, as stated in Moore's law.
    - We have to constrain the number of transistors that are switching to save dynamic power consumption. So only a tiny fraction of an integrated circuit is active; the remaining inactive area is dark silicon.
    - We sometimes apply a power-saving strategy(Race-to-dark) where cores run at their highest operating frequency to enter the lowest operating frequency state as soon as possible, saving static and dynamic power.

💡 Comments:

Question 2 (y2017-p5-q1)

- (a) Write one or more lines of SystemVerilog that correspond to a complete module for each of the following circuits. Aim for simplicity and explain any subtleties of your implementation.

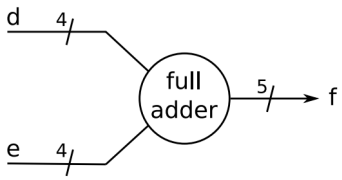
(i)



[4 marks]

```
module orgate(
  input logic [6:0] a, // 7-bit input
  output logic b); // 1-bit output

  always_comb b = !(a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6]);

endmodule
```

(ii)

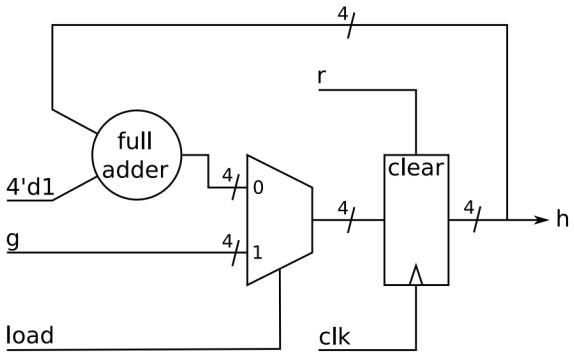


[4 marks]

```
// without using logic gates
module fourbit_fulladder(
  input logic [3:0] d;
  input logic [3:0] e;
  output logic [4:0] f);

  always_comb f = d + e;

endmodule
```

```
// using logic gates
module onebit_fulladder(
  input logic cin; // 1-bit carry in
  input logic a; // 1-bit input
  input logic b; // 1-bit input
  output logic cout; // 1-bit carry out
  output logic s); // 1-bit output

  XOR3 myxor(cin, a, b, s); // s = cin xor a xor b
  assign cout = (a & b) | (a & cin) | (b & cin);

endmodule

module fourbit_fulladder(
  input logic [3:0] d;
  input logic [3:0] e;
  output logic [4:0] f);

  wire tc0, tc1, tc2;
  onebit_fulladder(0, d[0], e[0], tc0, f[0]);
  onebit_fulladder(tc0, d[1], e[1], tc1, f[1]);
  onebit_fulladder(tc1, d[2], e[2], tc2, f[2]);
  onebit_fulladder(tc2, d[3], e[3], f[4], f[3]);

endmodule
```

(iii)



$[4 \text{ marks}]$

```
module circuit3(
  input logic [3:0] n;
  input logic [3:0] g;
  input logic load;
  output logic [3:0] h);

  always_ff @(posedge clk or posege r)
    if (r)
      h <= 0;
    else
      begin
      if (load)
        h <= g;
      else
        fourbit_fulladder(n, h, h);
      end
```

```
endmodule
```

- (b) Consider the following SystemVerilog module:

```
module gcd(
           input  logic         clk,
           input  logic         rst,
           input  logic         start,
           input  logic [15:0]  Ain,
           input  logic [15:0]  Bin,
           output logic [15:0]  answer,
           output logic         done
           );

   logic [15:0] a, b;
   always_ff @(posedge clk or posedge rst)
     if(rst)
       begin
          a <= 0;
          b <= 0;
          answer <= 0;
          done <= 0;
       end
     else
       if(start)
         begin
            a <= Ain;
            b <= Bin;
            done <= 1'b0;
         end
       else if(b==0)
         begin
            answer <= a;
            done <= 1'b1;
         end
       else if(a>b)
         a <= a-b;
       else
         b <= b-a;
endmodule
```

  - (i) For inputs Ain=21 and Bin=15, complete the following state transition table (where X means undefined or unknown values): [6]

input			current state		next state			
Ain	Bin	start	a	b	a'	b'	done'	answer'
21	15	1	X	X	21	15	0	X
21	15	0	21	15				

GCD

Aa input(Ain, Bin, start)	Current state(a, b)	Next state(a', b', done, answer)
(21, 15, 1)	(X, X)	(21, 15, 0, X)
(21, 15, 0)	(21, 15)	(6, 15, 0, X)
(6, 15, 0)	(6, 15)	(6, 9, 0, X)

input(Ain, Bin, start)	Current state(a, b)	Next state(a', b', done, answer)
(6, 9, 0)	(6, 9)	(6, 3, 0, X)
(6, 3, 0)	(6, 3)	(3, 3, 0, X)
(3, 3, 0)	(3, 3)	(3, 0, 0, X)
(3, 0, 0)	(3, 0)	(3, 0, 1, 3)

- (ii) What values of Ain and Bin will cause the module to fail to terminate (i.e. done will never go high)? [2]
  - If $a = 0$ and b is a positive number, we will run into an infinite loop where a and b remain unchanged, so never terminated.
  - If either a or b is a negative number, then b will never go to 0.

💡 Comments:

Question 3 (y2016-p5-q1)

Below is a functionally and syntactically correct mysterious module written in SystemVerilog.

```
typedef enum { opNone, opIn, opOut } operationT;

module mystery
  # (parameter depth, parameter width)
   (
   input  clk,
   input  rst,
   input  operationT op,
   input  logic [width-1:0] dataIn,
   output logic [width-1:0] dataOut,
   output logic empty,
   output logic full,
   output logic error);

   logic [width-1:0] mem[depth-1:0];
   reg [$clog2(depth-1)+1:0] head;
   // where $clog2(x) = ceiling(log_base_2(x))

   always_comb
     begin
        full    = head>=depth;
        empty   = head==0;
        error   = ((op==opIn) && full) || ((op==opOut) && empty);
        dataOut = empty ? -1 : mem[head-1];
     end

   always @(posedge clk)
     if(rst)
       head <= 0;
     else
       if(!error)
         case(op)
           opIn: begin head <= head+1; mem[head] <= dataIn; end
           opOut:      head <= head-1;
         endcase // case (op)
endmodule
```

- (a) What is the function of the mystery module? Include in your answer the behaviour when the module is complete (full==1) or empty (empty==1). What does dataOut output? What does input op do? [8 marks]

    - Simulate a stack(Last In Last Out) and its corresponding operations, such as pushing into a stack and pop from it.
    - When the module is full, the error would be 1 if we want to add data into the stack (operation is `opIn`).
    - When the module is empty, the error would be 1 if we want to pop data from the stack (operation is `opOut`).
    - `dataOut` would output -1 if the stack is empty or output the peek/head element in the stack. If `opOut`, then data would be pop out and store in `dataOut`.
    - `op` distinguish between two operations - `opIn` inputs data into the stack, or `opOut` would pop the data out.
- (b) What is a production test, and how does it differ from the functional test? [2 marks]
    - Production test is a test after manufacture and focuses on the reliability of the product rather than its functionality.
    - Functional testing is a quality assurance test that makes sure the product is aligned with its specification.
- (c) What are the key challenges in functionally testing the mystery module? [5 marks]
    - We need to test every case or branch of the program, including pop from an empty stack, push into a full-stack, get the peek element or pop the peek element from the stack, etc.
    - There are many possible states for the program, so functional testing of the module is quite time-consuming.
    - Error handling and possible edge cases need to be taken care of to ensure the module is functionally correct.
    - The stack depth and data width is unknown and have many possibilities, so proper testing datasets need to be devised.
- (d) What are the challenges in undertaking a production test of the mystery module, and how do these challenges compare with those for the functional tests? [5 marks]
    - The protection test would need to ensure we can access every piece of data properly (e.g., contents of the head pointer, each bit in `mem[]`).
    - However, since we assumed the module is functionally correct while doing production testing, we could devise one test dataset to test on.
- (a*) What is SystemVerilog?
    - A hardware description language is used to model, design, simulate and test electronic systems.

    - It combines the features of Hardware description languages such as Verilog with features from C and C++(e.g., modularity), so much more readable and understandable, improves programming productivity.
- (b*) What is the metastability of D flip-flops, and how does it occur?
    - When external asynchronous inputs violate setup or hold time, an unstable signal leads to undefined behaviour and the D flip-flops output neither 0 nor 1.
    - The unbalanced output is meta-stability which breaks down digital abstraction. We could link two flip-flops synchronises to synchronise inputs and avoid metastability.
- (c*) Brief explanation: what are FPGAs, and what are their benefits?
    - It stands for Field Programmable Gate Array - an integrated circuit that a user can program after being manufactured. It contains several reconfigurable components, including look-up tables (LUT), adaptive logic modules (ALM), logic array block (LAB) and hierarchical programming wiring etc.
    - Benefits:
        - Independence of component manufacturer. Because the functionality is not in the module itself but the configuration on the user side.
        - Fully programmable, updates and adaptions can be carried out even after delivery, so improved productivity.
        - Fast and efficient systems offer the possibility of developing plans that are precisely tailored to the intended task.
        - Ideal for real-time applications because the offered flexibility enabled complicated calculations to be carried out concisely.
        - Massive parallel data processing, FPGA offers a scalable solution via the parallel processing of data.

> 💡 Comments: I haven't heard of the production and functional tests and cannot find them in the slides, so I searched online for these two terms.

Question 4

Describe the steps that transform a program written in a high-level language such as C into a representation directly executed by a computer processor.

- Codes written in a high-level language are fed into the compiler to compile to codes in assembly language, assembly codes are texture representations of instructions
- The assembler would convert the assembly codes into binary programs including encoded instructions and data and are readable by a computer processor

> 💡 Comments:

Question 5

Discuss essential terms relevant to the performance of processors.

- Since processor performance (overall running programs) can be computed as follows.

$$\text{performance} = \frac{1}{\text{CPI time}} = \frac{1}{\text{instruction counts} \times \frac{\text{Clock cycles}}{\text{instruction counts}} \times \frac{\text{Seconds}}{\text{Clock cycles}}}$$

  - Instruction counts depend on the programs, the compiler optimisation and whether the instruction set is a RISC or CISC.
  - Clock cycles per instructions (CPI) depend on the programs, the compiler optimisation and the type of instruction set, and the processor design. The average CPI is affected by the instruction mix of all programs.
  - Clock rate depends on the instruction set architecture, the processor design and the implementation technology.
- Performance via parallelism:
  - Instruction-level parallelism where we execute more than one instruction in the same clock cycle.
  - Thread-level parallelism where N sequential threads run on N processors or N-multithreaded processors.
  - Data-level parallelism, where instruction can act on more than one data, is widely used in GPU.
- Performance via pipelining:
  - A production line for executing instructions, while one instruction is being performed, another is being decoded etc.
  - RISC has a limited instruction set; it fits the compiler and enables pipelining to improve the processor's efficiency.
- Performance via prediction:

- Predict what might happen next to make the pipeline and memory subsystem more efficient.
- Control-flow prediction to predict what instructions will be executed next.
- Data-value prediction to predict whether a value would change or not, or whether two pointers would alias to the same address
- Data-access pattern prediction to prefetch data that might be used shortly by exploiting data access's temporal and spatial statistical properties.

💡 Comments:

Question 6

Explain the term "Instruction Set Architecture - ISA", and why does ISA matter?

- Instruction set architecture is a repository of instructions or primitive operations that the processor can perform, including RISC and CISC.
- ISA directly affect the instruction counts of programs, the clock cycles per instruction (CPI) and the clock rate, so the choice of ISA influences the processor performance.
- Understand what ISA can do and how the compilers make use of those instructions can boost programming efficiency.

💡 Comments:

Question 7 (y2021-p5-q1)

- (a) Moore's law and Dennard scaling both predict scaling properties of CMOS chips. What are the differences between these predictions, and which predictions are valid today? [4 marks]
  - Moore's law predicts the number of transistors on a chip would double every two years.
  - Dennard scaling predicts that the power density stays the same with smaller transistors since the power consumption is in proportion to the area, so it will halve if the place halves.
  - Moore's law is still valid today, while Dennard scaling has broken down around 2006 because the physical limits of the transistors cause the chip to heat up and

increases static power consumption.

- (b) How is the critical path in a clocked digital CMOS circuit determined, and how does it impact the maximum clock frequency? [4 marks]
  - The critical path is the longest combinatorial path in a circuit; it determines whether the maximum clock frequency is safe or not.
  - We first follow the same constraints information as in the place and route stage, including clock frequencies, to ensure that digital abstraction is valid.
  - Then also need to consider the physical characteristics of the circuits as temperature changes and statistical variability in performance.
- (c) What is a function calling convention, and how does it impact the RISC-V instruction set architecture (ISA) design? [4 marks]
  - Assembly codes are based on conventions; function calling conventions include registers, function calls, and entering and existing functions; following these would reduce the complexity of debugging.
  - It helps to ensure that we do not need to store all values; for example, only the values in the `s` registers would need to be stored, while `t` and `a` register values do not. `x0` being zero is because zero is very useful, and holding a register would boost efficiency.
  - Registers:
    - `ra` holds the return address
    - `sp` contains the current base of the stack, which goes downwards
    - `t` registers stores local variables
    - `s` registers stores global variables
    - `x0` always keep the value zero
    - `a` registers have function arguments
  - Function calls:
    - `jal` instruction to a label (`jal ra label`)
    - `jalr` instruction to a register `rd` (`jalr ra rd imm`)
  - Entering & existing functions:
    - `sp` and `s` will have the same values after exiting the functions
    - We would return to the address stored in `ra`
- (d) Consider the following C function that computes the greatest common divisor and the assembler produced by the compiler. The assembler has been split into segments.

Describe what function each segment performs. [8 marks]

```
int gcd(int n1, int n2) {
  if (n2 == 0)
    return n1;
  else
    return gcd(n2, n1 % n2);
}

##### Segment A #####
gcd:
        bne     a1,zero,.L7
        jr      ra
##### Segment B #####
.L7:
        addi    sp,sp,-16
        sw      ra,12(sp)
##### Segment C #####
        mv      a5,a1
        rem     a1,a0,a1
        mv      a0,a5
        jalr    ra, gcd
##### Segment D #####
        lw      ra,12(sp)
        addi    sp,sp,16
        jr      ra
```

- Segment A:
  - Corresponds to the first `if` condition loop.
  - a_1 stores the value of n_2, `bne` checks the condition whether a_1 is not zero, if so, then jump to the loop `.L7`(the `else` condition loop), else return from the function, the returned value n_1 is stored in `ra`.
- Segment B:
  - Add -16 to stack pointer `sp` using add-immediate because we need to allocate a new call stack to the function `gcd`.
  - Store the value $12 + sp$ into `ra`.
- Segment C:
  - Store the value of a_1 into a_5 (`a5 = a1`)
  - Calculate the remainder $a_1 = a_0 \% a_1$(`a1 = a0 % a1`)
  - Store the value of a_5 into a_0 (`a0 = a5`)
  - Call the function `gcd` with a return address stored in `ra = pc + 4`
- Segment D:
  - load the data from memory location $12 + sp$ into `ra`
  - Add 16 to stack pointer `sp` using add-immediate. Because we want to return back to the previous call stack.
  - `jr ra` return from function with return value stored in `ra`.

💡 Comments:

Question 8

Summarise the main message from lesson 1 in 1-3 sentences?

- The von Neumann architecture and the memory hierarchy are still in use.
- Moore's law impacted the past processor trends but may end in the future because of the physical limits of transistors and imposed challenges.
- Dennard scaling indicates processor performance trends when combined with Moore's law, but it breaks down after 2006 because of power consumption challenges.

💡 Comments:

Question 9

Summarise the main message from lesson 2 in 1-3 sentences?

- Designers need to meet the timing requirements and preserve digital abstraction throughout the stages of clocked digital design (e.g., synthesis & automatic optimisation, place & route, timing analysis and simulations).
- The HDL simulation can check the design's high level while the gate-level simulation is a more accurate low-level simulation.
- The FPGA is used for low to medium volume production, while the ASIC is an entire custom chip for large-scale volumes.

💡 Comments:

Question 10

Summarise the main message from lesson 3 in 1-3 sentences?

- The processor performance is affected by the programs, the compiler optimisation, the instruction set architecture, the processor design, and implementation technology.
- The processor performance can be improved via parallelism, pipelining and prediction.

- We use high-level languages, a hierarchical memory structure and redundancy designs to maximising productivity, efficiency and reliability.

💡 Comments:

Question 11

Summarise the main message from lesson 4 in 1-3 sentences?

- RISC ISA aims to target the compiler, make the typical case fast, enable the independence of operations with registers, and simplify the instruction decoding process.
- All RISC-V variants are based on a 32-bit instruction format but different register width and virtual address space(e.g., RV64I has a 64-bit register width and 64-bit address space).

💡 Comments: