NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

EXAMINATION FOR

Semester 1 AY2011/12

CS2100 – COMPUTER ORGANISATION

Nov 2011 **Time allowed: 2 hours**

Your Matriculation Number:

INSTRUCTIONS TO CANDIDATES

1. This examination paper consists of **SIX (6)** questions and comprises **TWENTY (20)** printed pages including this page.
2. This is an **OPEN BOOK** examination. You may use any approved calculators but not any PDA or laptop, especially those capable of external connectivity or communication.
3. Answer *all* questions. Note that the full mark for each question is different.
4. Write your answers on *this* **QUESTION AND ANSWER SCRIPT**. Answer only in the space given. Any writing outside this space will not be considered. No other submission is allowed.
5. Fill in your Matriculation Number with a pen, clearly on every page of this QUESTION AND ANSWER SCRIPT.
6. You may use pencil to write your answers.
7. At the end of the examination, please check to ensure that your script has all the pages properly stapled together.
8. Note that when a number is written as “0xNNNN” it means that “NNNN” is in base 16.

Total Score /100

QUESTION 1

(1a) For the following 3-bit standard Gray code, fill in the *even* parity bit.

ANSWER:

A	*B*	*C*	Even Parity (*P*)
0	0	0	**0**
0	0	1	**1**
0	1	1	**0**
0	1	0	**1**
1	1	0	**0**
1	1	1	**1**
1	0	1	**0**
1	0	0	**1**

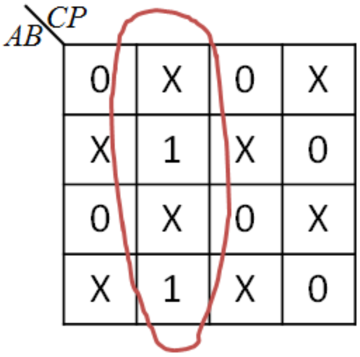
(1b) You are to design a 4-bit synchronous counter that counts the above 4-bit sequence looping back to 000 after reaching 100, using two T-flip-flops (for b_2 and b_0) and two JK flip-flops (for b_1 and P). First, fill in the control signals in the table below.

ANSWER:

Current State				Next State				T_A	J_B	K_B	T_C	J_P	K_P
A	*B*	*C*	*P*	*A*	*B*	*C*	*P*						
0	0	0	0	0	0	1	1	**0**	**0**	**X**	**1**	**1**	**X**
0	0	1	1	0	1	1	0	**0**	**1**	**X**	**0**	**X**	**1**
0	1	1	0	0	1	0	1	**0**	**X**	**0**	**1**	**1**	**X**
0	1	0	1	1	1	0	0	**1**	**X**	**0**	**0**	**X**	**1**
1	1	0	0	1	1	1	1	**0**	**X**	**0**	**1**	**1**	**X**
1	1	1	1	1	0	1	0	**0**	**X**	**1**	**0**	**X**	**1**
1	0	1	0	1	0	0	1	**0**	**0**	**X**	**1**	**1**	**X**
1	0	0	1	0	0	0	0	**1**	**0**	**X**	**0**	**X**	**1**

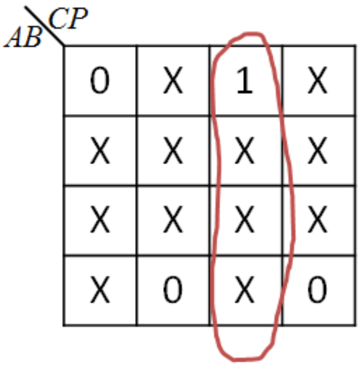
(1c) Draw a K-map for each of the control signal and give the minimum product-of-sum expression for each.

ANSWER:



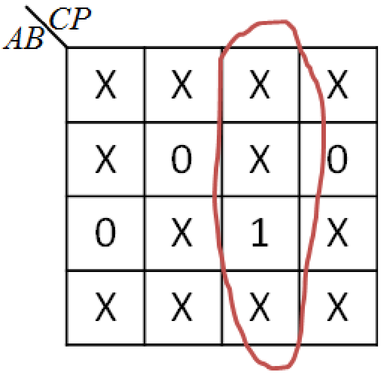
0	X	0	X
X	1	X	0
0	X	0	X
X	1	X	0

$T_A = C'\cdot P$



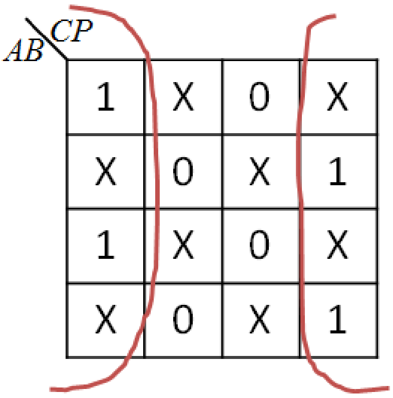
0	X	1	X
X	X	X	X
X	X	X	X
X	0	X	0

$J_B = C\cdot P$



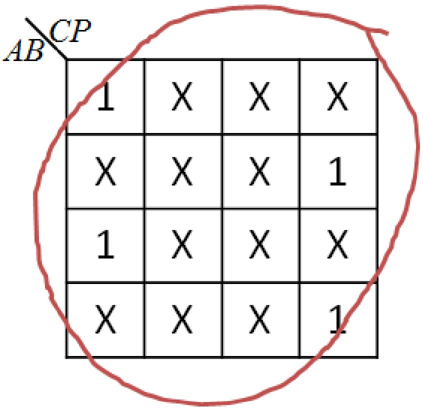
X	X	X	X
X	0	X	0
0	X	1	X
X	X	X	X

$K_B = C\cdot P$



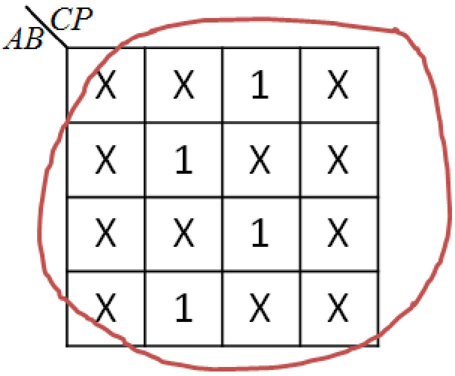
1	X	0	X
X	0	X	1
1	X	0	X
X	0	X	1

$T_C = P'$



1	X	X	X
X	X	X	1
1	X	X	X
X	X	X	1

$J_P = 1$

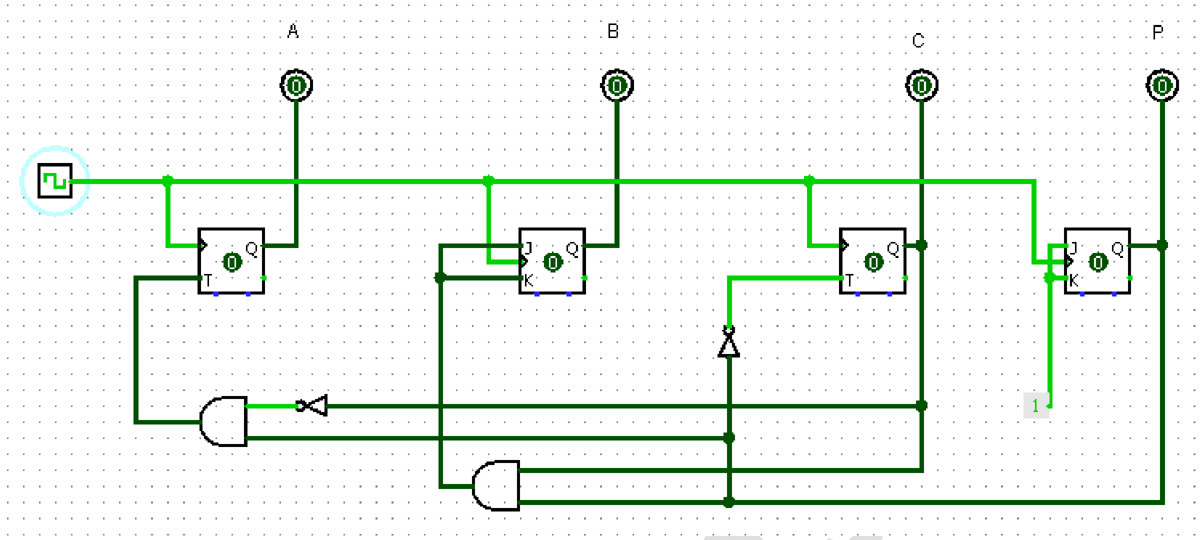


X	X	1	X
X	1	X	X
X	X	1	X
X	1	X	X

$K_P = 1$

(1c) Draw the final circuit for the counter.

ANSWER:



QUESTION 2

(2a) What is the MIPS instruction that is encoded by the hexadecimal number **`0x2acaff85`**?

ANSWER:

```
slti $10, $22, -123
```

or

```
slti $t2, $s6, -123
```

(2b) Encode the MIPS instruction “**`xor $v0, $a1, $t9`**”, leaving your answer as an 8-hexadecimal digit number.

ANSWER:

```
0x00b91026
```

(2c) Consider the following C function:

```
char *ToUpper(char *s1, char *s2);
```

It will copy the C string pointed to by **`s1`** to **`s2`** and convert all lower case letters to upper case ones. In other words, at the end of its execution, **`s2`** will be a copy of **`s1`** except that all lower case letters will be converted to upper case ones. All other characters remain unchanged. The function returns **`s2`** as the result. Write an *efficient* MIPS assembly routine for this function. Note that the ASCII code for ‘A’ (uppercase A) is 0x41, while the ASCII code for ‘a’ (lowercase a) is 0x61.

ANSWER:

```
ToUpper:
       addi   $v0, $a1, 0            # return value of s2

Loop:
       lb     $t2, 0($a0)            # *s1
       beq    $t2, 0, finish         # end of string s1?
       slti   $t3, $t2, 0x61         # < ‘a’?
       bne    $t3, $0, continue
       slti   $t3, $t2, 0x7A         # >= ‘z’ which is !(< ‘z’)
       beq    $t3, $0, continue
       addi   $t2, $t2, -0x20

continue:
       sb     $t2, 0($a1)            # store into s2
       addi   $a0, $a0, 1            # incr address of s1
       addi   $a1, $a1, 1            # incr address of s2
       j      Loop

finish:
       sb     $0, 0($a1)             # end of string char
       jr     $ra
```

QUESTION 3

(3a) What is the number represented by the IEEE Standard 754 single precision floating point number **`0xBDD40000`**? Leave your answer in decimal scientific notation.

ANSWER:

-1.035156×10^{-1}

(3b) What is the *smallest* and the *biggest* gap between two *adjacent* (consecutive representable) normalized IEEE Standard 754 single precision floating point numbers?

ANSWER:

Smallest gap is between the number with sign = 0, exponent = 00000001_2, fraction = $1.00000000000000000000000_2$, and sign = 0, exponent = 00000001_2, fraction = $1.00000000000000000000001_2$ In other words, it is $1\times 2^{-23} \times 2^{-126} = 2^{-149} \approx 1.40123 \times 10^{-45}$.

Largest gap is between the number with sign = 0, exponent = 11111110_2, fraction = $1.11111111111111111111110_2$, and sign = 0, exponent = 11111110_2, fraction = $1.11111111111111111111111_2$ In other words, it is $1\times 2^{-23} \times 2^{127} = 2^{-104} \approx 2.02824 \times 10^{31}$.

(3c) Suppose an internal computation has resulted in the following floating point number:

Sign = 1

Exponent = 01100101_2

Fraction bits = $10001111111111111100110_2$

Guard bit = 1

Round bit = 0

Sticky bit = 1

What is the IEEE standard 754 single precision format number (leave your answer as a hexadecimal string) after (i) rounding to nearest, (ii) rounding to zero, and (iii) rounding to +∞?

ANSWER:

(i) Round to nearest:

Sign = 1

Exponent = 01100101_2

Fraction bits = $10001111111111111100111_2$

Result = 0xB2C7FFE7

(i) Round to zero:

Sign = 1

Exponent = 01100101_2

Fraction bits = $10001111111111111100110_2$

Result = 0xB2C7FFE6

(i) Round to +∞:

Sign = 1

Exponent = 01100101_2

Fraction bits = $10001111111111111100110_2$

Result = 0xB2C7FFE6

(3d) The associativity rule is not observed in floating point arithmetic. For example, in general, $(A + B) + C \neq A + (B + C)$ where A, B, and C are IEEE Standard 754 single precision floating point numbers. Give an example of A, B, and C that will illustrate this property. State clearly which rounding mode you are assuming in your example.

ANSWER:

There are many possible examples. Below are three such normalized numbers:

A:

Sign = 0

Exponent = 01111111_2

Fraction bits = $00000000000000000000000_2$

B:

Sign = 0

Exponent = 01100111_2

Fraction bits = $00000000000000000000000_2$

C:

Sign = 0

Exponent = 01100111_2

Fraction bits = $00000000000000000000000_2$

The number A is +1. Both B and C are the number $1.0_2 \times 2^{-24}$. In the addition of (A + B), After denormalization shifts, the bits of B will start at the Guard bit. If we assume Round-To-Nearest, since the Guard bit is 1 but the Round and Sticky bits are 0, and with the LSB of the largest number being 0, after rounding, the result will be exactly *A*. When this is result is now added to *C*, the same will happen.

However, if (*B* + *C*) is done first, then the intermediary result would be $1.0_2 \times 2^{-23}$. This will cause a 1 to appear at the least significant bit position. Hence the result of adding this to *A* will be:

Sign = 0

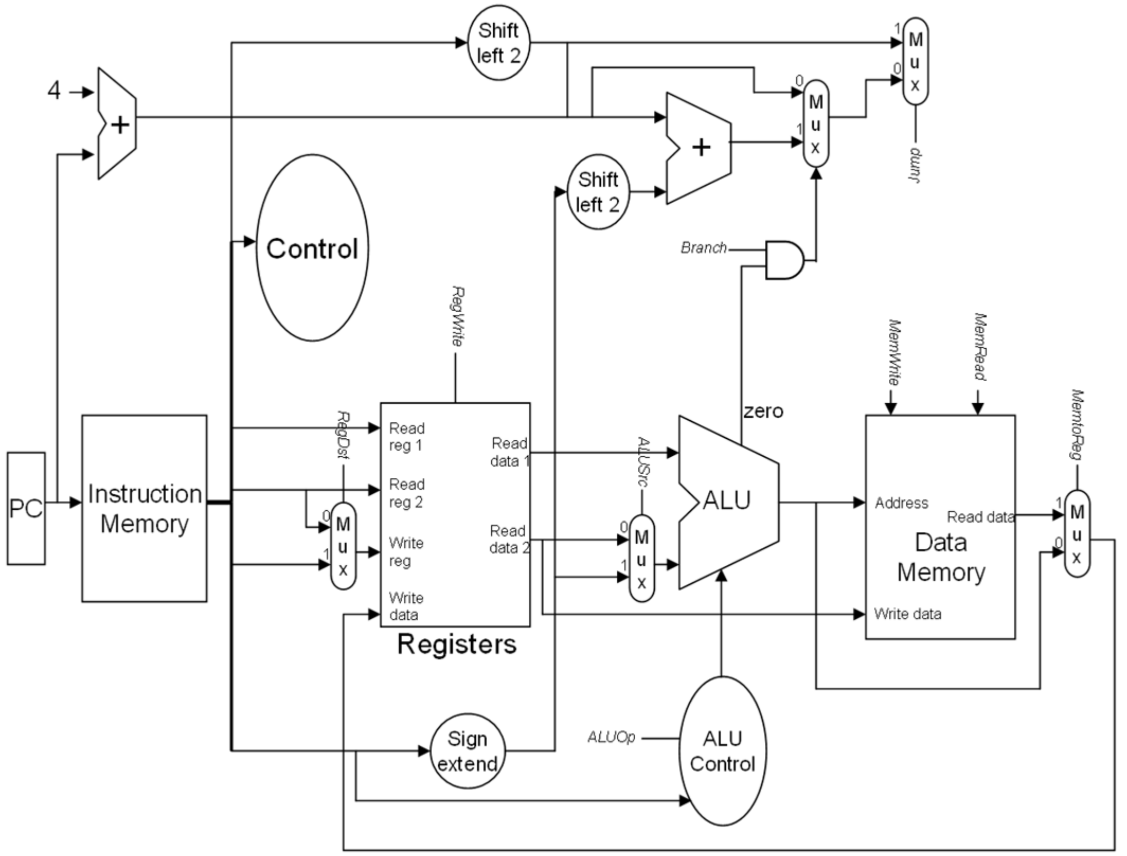
Exponent = 01111111_2

Fraction bits = $00000000000000000000001_2$

which is not *A*, and hence different from the result of the first order of additions.

QUESTION 4

(4a) The diagram below is a simplified version of the single cycle datapath.



Write down the control signals that would constitute the execution of the instruction:

```
addi  $s0, $t5, -1
```

If any of the values are more than 1 bit, write down the hexadecimal value. You may also use ‘X’ for “don’t care” or unknown values. However, assume that the contents of the both registers mentioned in the instruction above are zeroes.

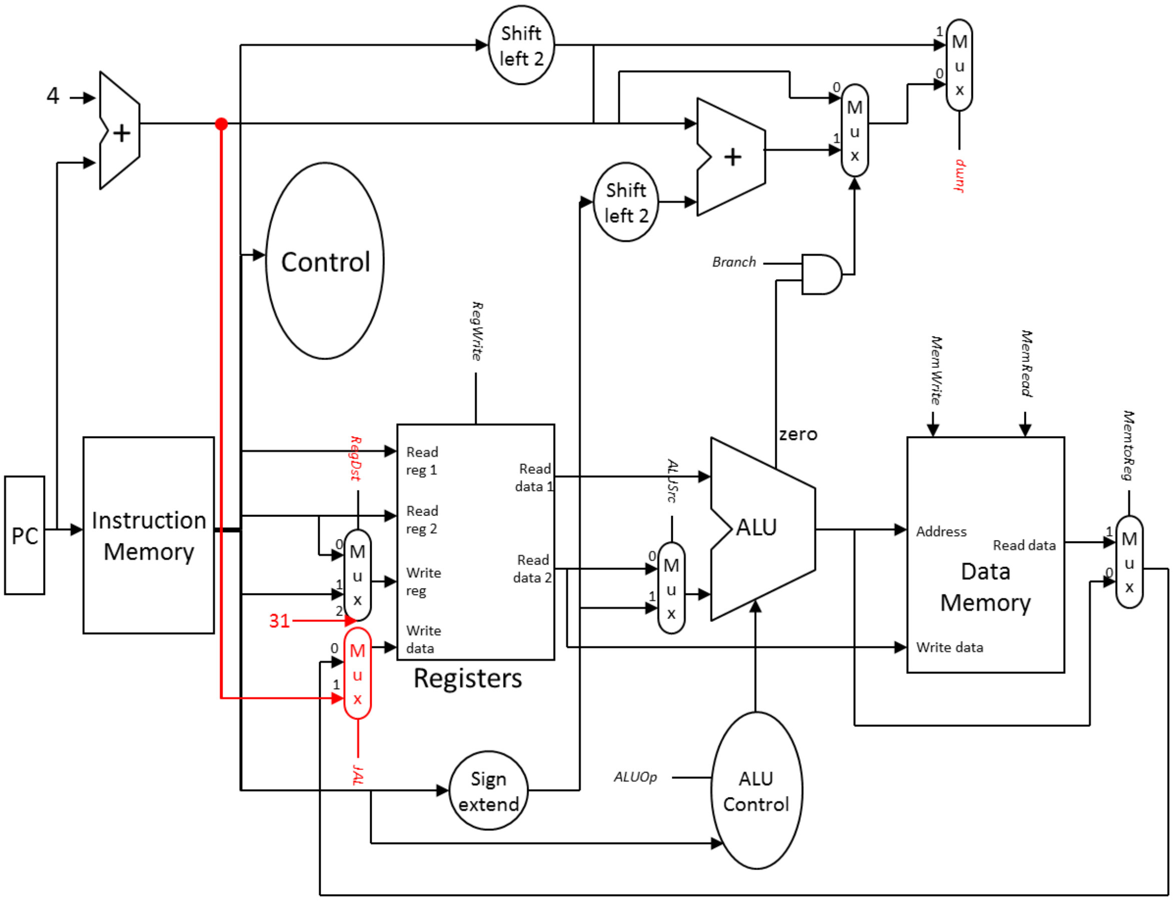
ANSWER:

Signal	**Value**
Read reg 1	0xD
Read reg 2	0x10
Write reg	0x10
RegDst	0
RegWrite	1
ALUSrc	1
Branch	0
Address	0xFFFFFFFF
ALUOp	0x2

MemWrite	0
MemRead	0
MemtoReg	0
Jump	0

(4b) By drawing additional components and signal lines, including possibly new control signals (which you should define), show how support for the **`jal`** ("jump and link") instruction can be added. The **`jal`** instruction is a jump instruction that also writes `PC`+4 into register **`$31`**. Besides drawing new lines and control signals on the diagram, you should also identify any changes to existing control or data signals, and how the instruction is implemented.

ANSWER:



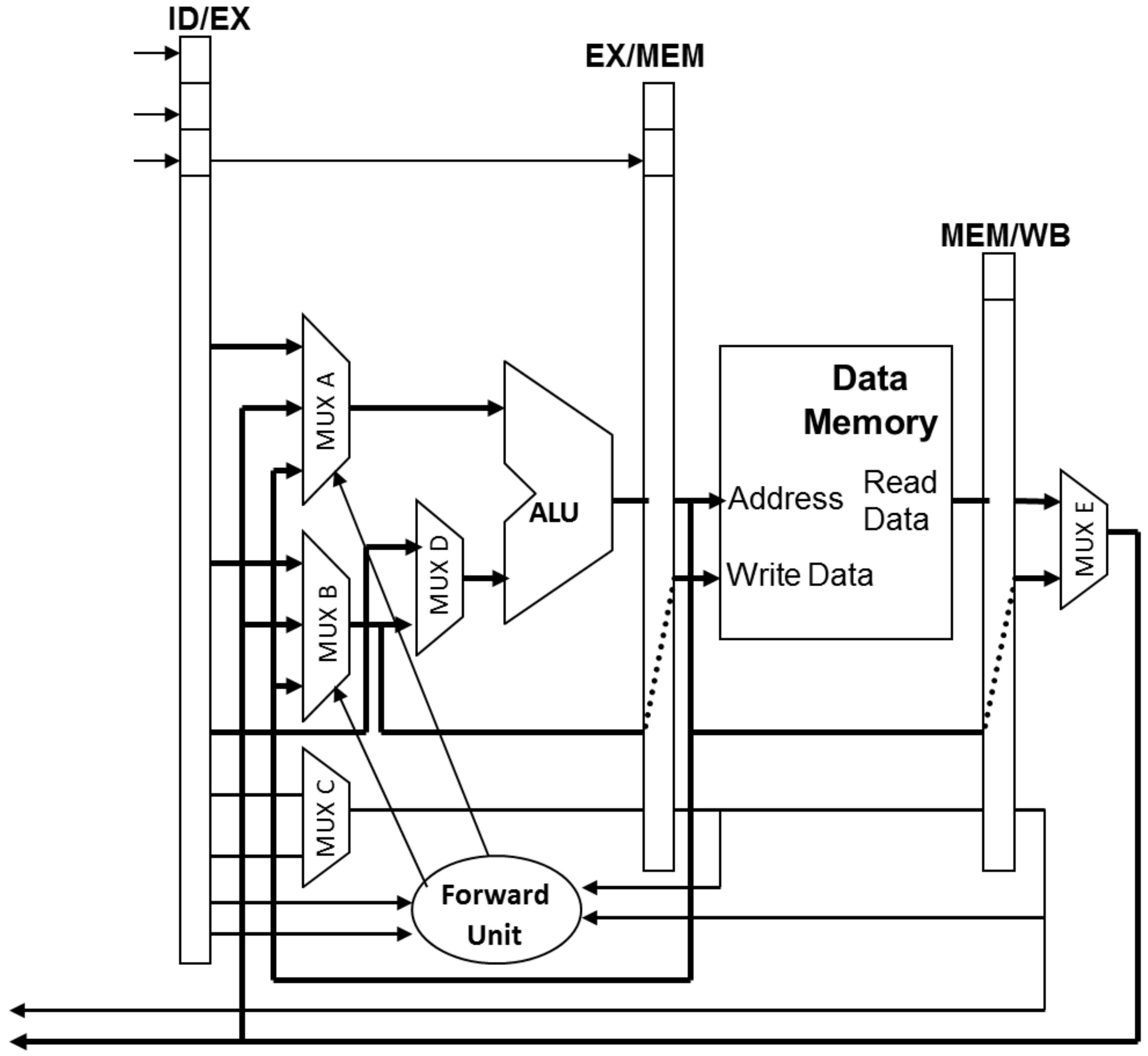
A hardwired number '31' is fed into an extended multiplexor guarded by the `RegDst` control signal. The `RegDst` signal needs to be modified so that for a `jal` instruction, `RegDst` needs to select input 2 ('31') as the write register address. The next change is that PC+4 is sent to a new multiplexor that guards the data to be written into the register file. This is guarded by a new `JAL` signal that is 1 for `jal` instructions, and 0 otherwise. Together, they will ensure that PC+4 is written into `$ra` (register 31) on a `jal` instruction. The `Jump` signal is also modified to be active on a **`jal`** besides a **`j`** instruction.

(4c) The diagram below is for the last three pipeline stages for a pipeline with forwarding facilities. Now consider the following two instructions:

```
add $1,$2,$3
sw  $1,4($3)
```

Describe how forwarding is done as these two instructions execute in these last three stages of the pipeline. You may want to annotate critical points in the diagram (say by drawing '①', '②', etc. on the diagram) so as to better explain the flow of data and the how the multiplexors implement that flow in each clock cycle. (5 marks)

ANSWER:



When the add instruction completes the EX and enters the MEM stage, its result will be available for forwarding. At this time, the sw instruction would be at the EX stage. The first port of the ALU will be the rs register value of the sw instruction (so no forwarding there, just the first input of MUX A will be pass along to the ALU), and the second input will be the immediate value (the first input of MUX D).

The result of the add instruction will enter MUX B as its third input. And the forwarding unit will select this as the output of MUX B. However, this does not make it into the second input of the ALU (as guarded by MUX D). Instead, it will be passed along, and in the next cycle becomes the write data to the data memory.

During the next cycle, the address computed by adding the rs register value of the sw together with the signed extended immediate will become the address, and the forwarded result of the add instruction will become the write data, thereby completing the successful execution of the sw instruction.

(4d) "Pipelining is one of the most important innovations in the architecture of processors." Explain the effects of pipelining as well as some of the drawbacks. (5 marks)

ANSWER:

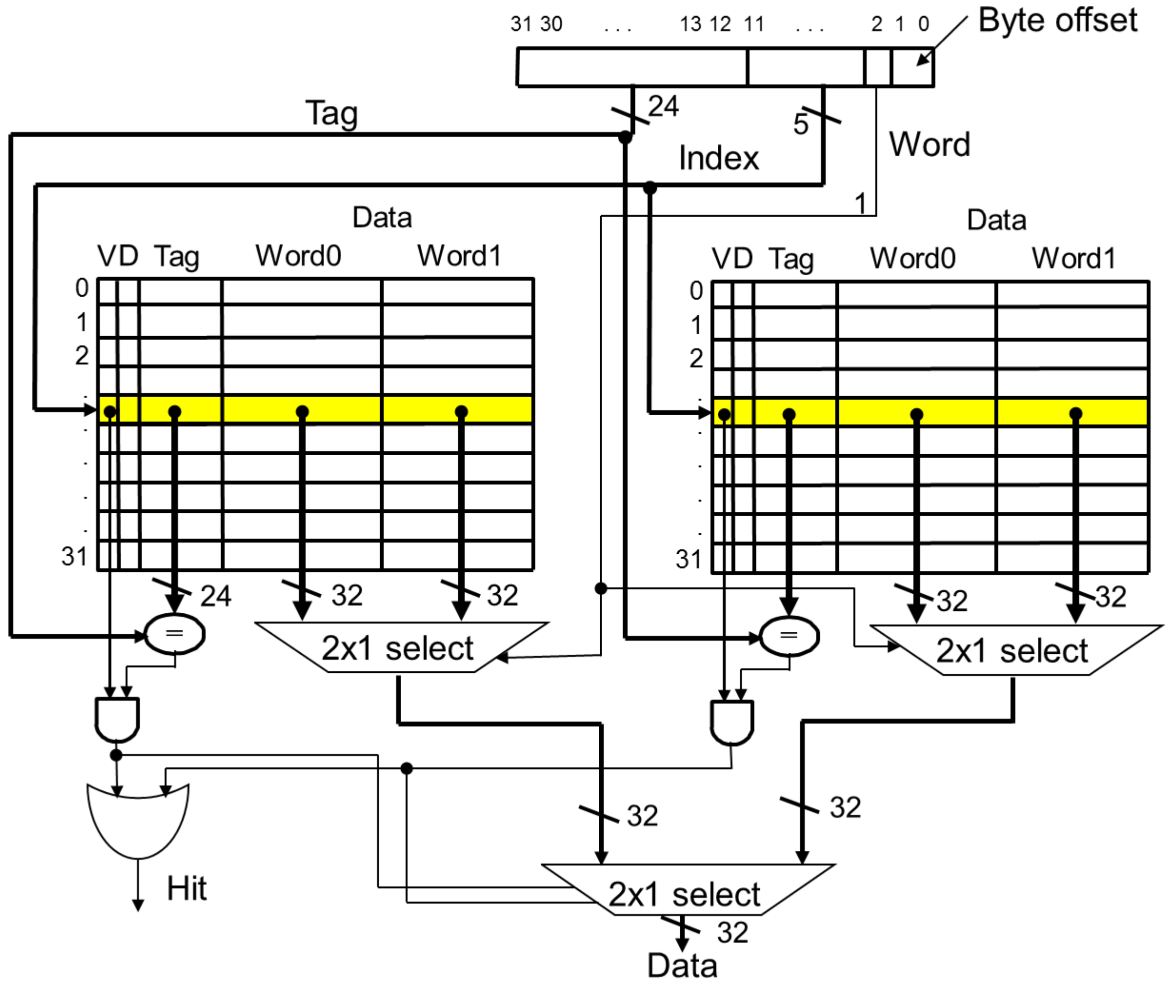
By splitting the processing of an instruction into finer stages, it is possible to start the execution of one instruction as soon as the instruction ahead has finished executing the first stage. This allows for several instructions to be concurrent processed inside the processor, thus improving throughput. However, the concurrent execution of instructions introduces hazards, namely, structural, control and data hazards arising from various dependences. This requires the assistance of extra hardware so as to ensure that the flow inside the pipeline is least disturbed. The addition of pipeline registers also introduces overhead in the processing of a single instruction.

Furthermore, because stages are more smaller, it is possible to use faster clocks in pipelined processors. This further improves throughput and performance.

QUESTION 5

(5a) Show the block diagram implementation for a 2-way set associative, 2-word per block, 1Kbyte write-back cache on a byte-addressable processor. A 'word' here is 4 bytes. You should also show clearly how the 32-bit address is decomposed for addressing the cache.

ANSWER:



(5b) Propose a C/Java data structure for simulating this cache. You may assume that the type 'long' holds 4 bytes.

ANSWER:

Total number of sets = 1024 / (2 * 4 * 4) = 32 sets.

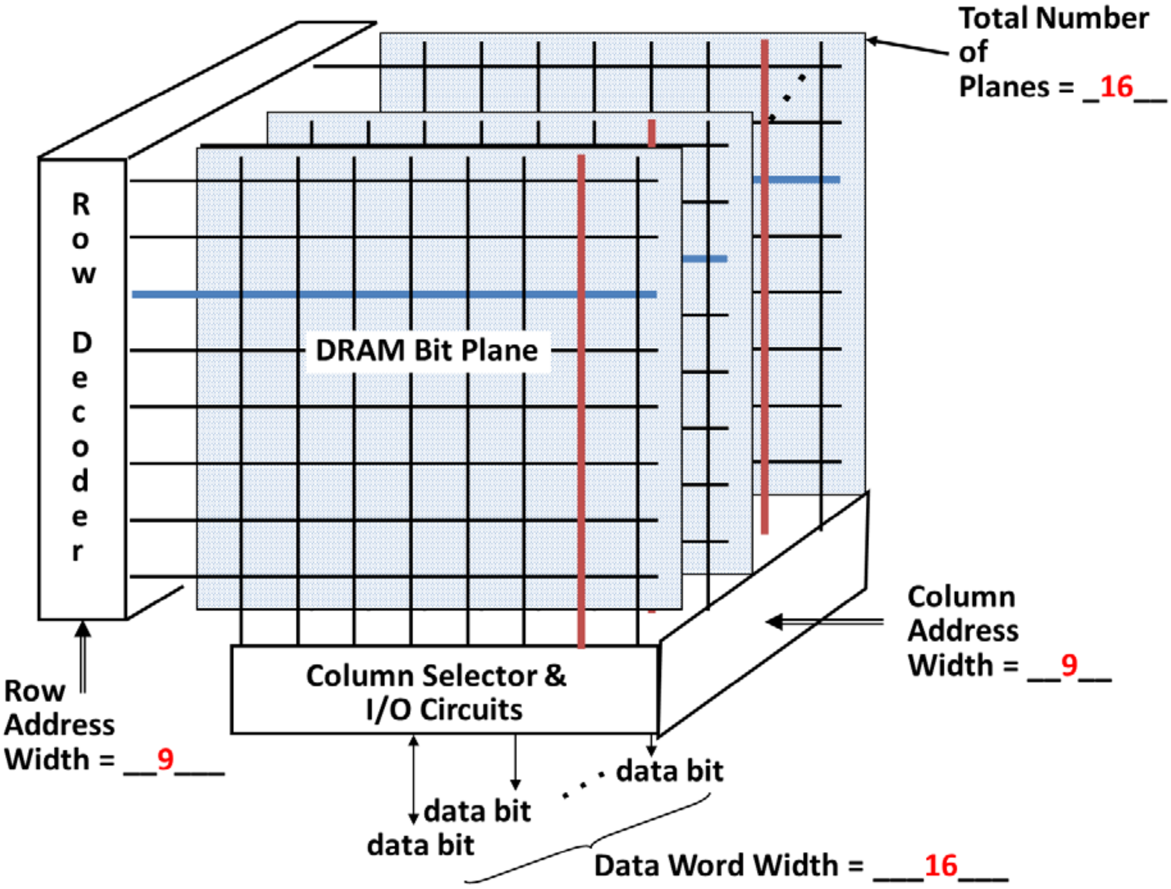
```
struct CACHE_BLK {

    struct WAY {
      long tag;
      bool valid;
      bool dirty;
      char data[2][4];
```

```
        } way[2];
    } set[32];
```

QUESTION 6

(6a) Using square DRAM bit planes of 512x512 bits, show how a 16 bit word DRAM organization would be like by filling in the details in the diagram below.

ANSWER:


Total Number of Planes = _16__
Row Decoder
DRAM Bit Plane
Column Address Width = __9__
Row Address Width = __9___
Column Selector & I/O Circuits
data bit
data bit
data bit
data bit
Data Word Width = ___16___

(6b) What is the size of the above implementation?

ANSWER:

16 planes will be used to realize a word. Hence the total size is 512 × 512 × 16 bits = 4194304 bits or 512Kbytes.

(6c) Assume that a 32-bit address machine has a page size of 4,096 bytes, and a 4 entry TLB. The machine runs sillyOS that has a fixed page allocation algorithm: any virtual page is mapped to a physical page that has the virtual page number added with 100_{10}. The machine boots up and encounters four memory addresses:

```
0x10031023
0x3F291367
0x8888DEAD
0x0001671E
```

What would be the entries in the TLB (ignoring the dirty, ref, and permission bits) after the page faults for these four addresses have been successfully serviced.

ANSWER:

Valid Bit	Virtual Page Number	Physical Page Number
1	0x10031	0x10095
1	0x3f291	0x3f2f5
1	0x8888d	0x888f1
1	0x16	0x7a

(6d) Why is the page allocation algorithm of sillyOS a very bad idea?

ANSWER:

The actual size of physical memory available will be much smaller than the virtual memory. There will not be sufficient physical pages to handle the demands of processes. Furthermore, this fixed allocation will result in a collision for virtual pages from two processes having the same virtual page numbers.

=== END OF PAPER ===