**Question #1 (4 points)**

Let focus on the Reorder Buffer (ROB) existing in the architecture of some superscalar processors.

You are requested to

1. Explain the role of the ROB (when an entry is allocated in it, when it is written, when it is read, when it is de-allocated)
2. Describe the ROB architecture, detailing the fields composing each entry
3. Summarizing the advantages stemming from the adoption of the ROB.

**Write here your answer**

**Question #2 (4 points)**

Let consider a MIPS64 architecture including the following functional units (for each unit the number of clock periods to complete one instruction is reported):

* Integer ALU: 1 clock period
* Data memory: 1 clock period
* FP arithmetic unit: 2 clock periods (pipelined)
* FP multiplier unit: 6 clock periods (pipelined)
* FP divider unit: 8 clock periods (unpipelined)

You should also assume that

* The branch delay slot corresponds to 1 clock cycle, and the branch delay slot is not enabled
* Data forwarding is enabled
* The EXE phase can be completed out-of-order.

You should consider the following code fragment and, **filling the following tables**, determine the pipeline behavior in each clock period, as well as the total number of clock periods required to execute the fragment. The values of the constants k1 and k2 are written in f10 and f11 before the beginning of the code fragment.

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MIPS64 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; for (i = 0; i < 10; i++) {

; v4[i] = (v1[i]+v2[i])/(v3[i]\*k);

; }

|  |  |  |
| --- | --- | --- |
| .data | Comments | Clock cycles |
| v1: .double “10 values” |  |  |
| v2: .double “10 values” |  |  |
| V3: .double “10 values” |  |  |
|  |  |
|  |  |
| .text |  |  |
| main: daddui r1,r0,0 | r1← pointer |  |
| daddui r2,r0,10 | r2 <= 10 |  |
| loop: l.d f1,v1(r1) | f1 <= v1[i] |  |
| l.d f2,v2(r1) | f2 <= v2[i] |  |
| add.d f5, f1, f2 | f5 <= v1[i] +v2[i] |  |
| l.d f3,v3(r1) | f3 <= v2[i] |  |
| mul.d f6, f3, f10 | f6 <= f3\*k |  |
| div.d f4, f5, f6 | f4 <= (v1[i]+v2[i]) /(v3[i]\*k) |  |
| s.d f4,v4(r1) | v4[i] <= f4 |  |
| daddui r1,r1,8 | r1 <= r1 + 8 |  |
| daddi r2,r2,-1 | r2 <= r2 - 1 |  |
| bnez r2,loop |  |  |
| Halt |  |  |
| Total |  |  |

6+(27)\*10=276

AR2 MUL6 DIV8

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| main: daddui r1,r0,0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| daddui r2,r0,10 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| loop: l.d f1,v1(r1) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| l.d f2,v2(r1) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| add.d f5, f1, f2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| l.d f3,v3(r1) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| mul.d f6, f3, f10 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| div.d f4, f5, f6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| s.d f4,v4(r1) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| daddui r1,r1,8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| daddi r2,r2,-1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| bnez r2,loop |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Halt |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Question #3 (6 points)**

Given a 8 x 5 matrix of bytes SOURCE representing unsigned numbers, write a 8086 assembly program which computes on 16 bits (two’s complement) the addition of all cells with indexes (i,j) where i+j is an even value, minus all the cells whose i+j is an odd value. Please consider that i ranges from 0 to 7 and j ranges from 0 to 4.

Please add significant comments to the code and instructions.

Friendly advice: before starting to write down the code, think at a possible (very) simple algorithm! The choice of the algorithm highly influences the complexity and length of the code.

Example:

matrix SOURCE

1 2 3 4 5

6 7 8 9 0

9 8 7 6 5

4 3 2 1 0

7 7 7 7 7

3 5 7 9 0

8 7 6 5 4

9 9 9 3 2

the cells with i+j even are added up, while the cells with i+j odd are subtracted

1+3+5+7+9+9+ …

-2-4-6-8-0-….

The result will be clearly on 16 bits in two’s complement.

**Write your code in a file saved in the 8086 folder.**

**Question #4 (9 points)**

The IEEE-754 SP standard expresses floating-point numbers in 32 bits:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 | 30 | 23 | 22 | 0 |
| sign | exponent | | mantissa | |

Bit 31 is 0 if the number is positive, 1 if negative.

Write the addFPpositiveNumbers subroutine, which receives in input two 32-bit numbers, considers them as IEEE-754 SP floating point numbers, and returns their sum (in the same format). Bit 31 of the two input numbers is always 0 (i.e., the two numbers are positive).

In details, the subroutine implements the following steps:

1. take the mantissa of the two parameters
2. set bit 23 of the mantissa to 1
3. compare the two exponents. If they are equal, the exponent of the result is the same. If they are different:
   1. the exponent of the result is the highest one
   2. shift right the mantissa of the number with the lower exponent by as many position as the difference between the two exponents.
4. sum the two mantissas: this is the mantissa of the result. If bit 24 of the mantissa of the result is 1:
   1. shift right the mantissa of the result by one position
   2. increment the exponent of the result by one.
5. set bit 23 of the mantissa of the result to 0.
6. combine the mantissa and the exponent to get the final result.

Example: parameter1 = 0100 0010 0100 1011 0000 0000 0000 0000

parameter2 = 0100 0001 1010 0100 0000 0000 0000 0000 41B40000

1. mantissa1 = 0000 0000 0100 1011 0000 0000 0000 0000  
   mantissa2 = 0000 0000 0010 0100 0000 0000 0000 0000
2. mantissa1 = 0000 0000 1100 1011 0000 0000 0000 0000  
   mantissa2 = 0000 0000 1010 0100 0000 0000 0000 0000
3. exponent1 = 1000 0100  
   exponent2 = 1000 0011
   1. exponentResult = 1000 0100
   2. mantissa2 = 0000 0000 0101 0010 0000 0000 0000 0000
4. mantissaResult = 0000 0001 0001 1101 0000 0000 0000 0000
   1. mantissaResult = 0000 0000 1000 1110 1000 0000 0000 0000
   2. exponentResult = 1000 0101
5. mantissaResult = 0000 0000 0000 1110 1000 0000 0000 0000
6. result = 0100 0010 1000 1110 1000 0000 0000 0000

Click on the following links to open web pages with the ARM instruction set

[http://www.keil.com/support/man/docs/armasm](http://www.keil.com/support/man/docs/armasm" \t "_blank)

[https://developer.arm.com/documentation/ddi0337/e/Introduction/Instruction-set-summary?lang=en](https://developer.arm.com/documentation/ddi0337/e/Introduction/Instruction-set-summary?lang=en" \t "_blank)

Note: the assembly subroutine must comply with the ARM Architecture Procedure Call Standard (AAPCS) standard (about parameter passing, returned value, callee-saved registers).

**Create a new project with Keil inside the “template” directory and write your code there. The “template” directory contains the subdirectories “led” and “button” that you can add to your project if you need them.**

**Question #5 (4 points)**

Add a C file (e.g. sample.c) to the project created in the previous exercise.

Write here the main function (which needs to be called from the Reset handler).

Inside the main function, call the addFPpositiveNumbers subroutine, passing two floating-point numbers. If the result is zero or positive, switch on led 4 and switch off all other leds. If the result is negative, switch on led 5 and switch off all other leds.