



## Computer architectures

Exam 31/08/2021



**Iniziato** martedì, 31 agosto 2021

**Terminato** martedì, 31 agosto 2021

**Tempo impiegato** 2 ore



### Domanda 1

Completo

Punteggio max.: 4

Let consider the Branch Prediction Mechanism based on the Branch History Table (BHT).

You are requested to

1. Describe the hardware architecture of a BHT 
2. Describe the behavior of a BHT, detailing when the processor accesses it, and which information it gets from it 
3. Assuming that the processor uses 32 bit addresses, each instruction is 4 byte wide, and the BHT is composed of 8 entries, identify the final content of the BHT if
  - The BHT is initially empty (i.e., full of 0s, meaning NotTaken)
  - The following instructions are executed in sequence
    - addi r2, r2, #1 located at the address 0x00B20020
    - add r3, r3, r2 located at the address 0x00B20024
    - bne r2, r3, l1 located at the address 0x00B20028; the branch is taken, and the branch target address is 0x00A60050
    - addi r2, r2, #1 located at the address 0x00A60050
    - bez r2,l2 located at the address 0x00A60054; the branch is not taken
    - andi r5, r5, #1 located at the address 0x00A60058
    - bez r2,l3 located at the address 0x00A6005C; the branch is taken, and the branch target address is 0x00BB0020
    - add r1, r2, r3 located at the address 0x00BB0020
    - bez r1,l4 located at the address 0x00BB0024; the branch is taken, and the branch target address is 0x00A50050.
4. For the example in the previous point, determine which of the 4 branch instructions were correctly predicted, and which were mispredicted.

### Domanda 2

Completo

Punteggio max.: 4

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: 5 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]/k1))+v3[i]/k2;
; }

```

Code	Comments	Clock cycles
.data		
v1: .double "10 values"		
v2: .double "10 values"		
v3: .double "10 values"		
v4: .double "10 values"		
.text		
main: daddui r1,r0,0	r1 ← pointer	5
daddui r2,r0,10	r2 ≤ 10	1
loop: l.d f1,v1(r1)	f1 ≤ v1[i]	1
l.d f2,v2(r1)	f2 ≤ v2[i]	1
mul.d f3, f1, f2	f3 ≤ v1[i]*v2[i]	6
div.d f4, f3, f10	f4 ≤ v1[i]*v2[i]/k1	10
l.d f5,v3(r1)	f5 ≤ v3[i]	0
div.d f6, f5, f11	f6 ≤ v3[i]/k2	8
add.d f7,f4,f6	f7 ≤ v1[i]*v2[i]/k1 + v3[i]/k2	2
s.d f7,v4(r1)	v4[i] ≤ f7	1
daddui r1,r1,8	r1 ≤ r1 + 8	1
daddi r2,r2,-1	r2 ≤ r2 - 1	1
bnez r2,loop		2
halt		1
	total	

```

main:
daddui
r1,r0,0
daddui
r2,r0,10
loop:
l.d
f1,v1(r1)

```

```

l.d
f2,v2(r1)
mul.d
f3, f1, f2
div.d
f4, f3,
f10
l.d
f5,v3(r1)
div.d f6,
f5, f11
add.d
f7,f4,f6
s.d
f7,v4(r1)
daddui
r1,r1,8
daddi
r2,r2,-1
bnez
r2,loop

halt

```

### Domanda 3

Completo

Punteggio max.: 6

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.

---

#### Informazione

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

<http://www.keil.com/support/man/docs/armasm>

<https://developer.arm.com/documentation/ddi0337/e/Introduction/Instruction-set-summary?lang=en>

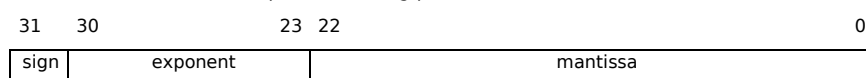
Note: Assembly subroutines must comply with the ARM Architecture Procedure Call Standard (AAPCS) standard (about parameter passing, returned value, callee-saved registers).

#### Domanda 4

Completo

Punteggio max.: 13

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



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

The exponent field ranges between 1 and 254, with a bias of 127. Consequently, the IEEE-754 SP standard can represent values ranging from  $2^{-126}$  to  $2^{128-1}$ .

However, the IEEE-754 SP standard can represent also values with an exponent lower than -126: they are called denormalized (or subnormal) numbers and are represented using leading zeros in the significand. Denormalized numbers are encoded with a biased exponent of 0, but are interpreted with the value of the smallest allowed exponent (i.e., 1).

Finally, exponent = 255 and mantissa = 0 represent the infinite value.

Write the `multiplyFPnumbers` subroutine, which receives in input two 32-bit numbers, interpreted as IEEE-754 SP floating point numbers, and returns their product

In details, the subroutine implements the following steps:

- For each input parameters:
  - if the exponent is higher than 0, the parameter is a normal number, with a leading 1. Therefore, set bit 23 of its mantissa to 1.
  - if the exponent is 0, the parameter is a denormalized number. Therefore, set its exponent equal to 1.
- The exponent of the result is: exponent of first parameter + exponent of second parameter - 126. If the exponent of the result is higher than 254, an overflow occurs: the procedure return infinity (i.e., exponent = 255, mantissa = 0).
- Multiply the mantissas of the two parameters by means of an unsigned 64-bit multiplication in order to have a 64-bit result.
- Logical shift left the 64-bit result by 8 positions.
- If the exponent of the result (computed at step 2) is negative or zero, then the result is a denormalized number. Take the first 32 bits (i.e., the most significant word) of previous 64-bit result. Logical shift them right by  $(1 - \text{exponent})$  positions: this is the mantissa of the result. Set the exponent of the result to 1.
  - Instead, if the exponent of the result is strictly positive, use a loop to logical shift the 64-bit result left by 1 position and decrement the exponent of the result as long as both the following conditions are true:
    - bit 23 of the mantissa is not set
    - the exponent of the result is higher than 1
- The mantissa of the result corresponds to the first 32 bits (i.e., the most significant word) after the shift.
- Check bit 23 of the mantissa of the result:
  - if it is set, clear it (the result is a normal value)
  - if it is not set, set the exponent to 0 (the result is a denormalized value)
- The sign of the result is 0 (positive) if the two parameters have the same sign, 1 (negative) otherwise
- Combine sign, exponent, and mantissa to get the final result.

Example: `parameter1 = 0100 0000 0101 0101 0101 0101 0101 0000`

`parameter2 = 1000 0000 0010 1010 1010 1010 1010 1000`

`sign1 = 0`

`exponent1 = 1000 0000`

`mantissa1 = 0000 0000 0101 0101 0101 0101 0101 0000`

`sign2 = 1`

`exponent2 = 0000 0000`

`mantissa2 = 0000 0000 0010 1010 1010 1010 1010 1000`

Note: the mantissas are represented in this example by putting 9 zeros in front of the 23 bits to arrive at the size of the registers (32bit), as subsequent operations involve a greater number of bits.

- `mantissa1 = 0000 0000 1101 0101 0101 0101 0101 0000`  
`exponent2 = 0000 0001`
  - `exponentResult = 1000 0000 + 0000 0001 - 0111 1110 = 0000 0011`
  - `mantissa1 * mantissa2 = 0000 0000 0000 0000 0010 0011 1000 1110 0011 0101 1100 0111 0001 1100 1000 0000`
  - after logical left shift, `mantissa1 * mantissa2 = 0000 0000 0010 0011 1000 1110 0011 0101 1100 0111 0001 1100 1000 0000 0000 0000`
  - The product `mantissa1*mantissa2` is logically shifted left by 2 positions (and the exponent is decremented accordingly). The mantissa of the result corresponds to the first word:  
`mantissaResult = 0000 0000 1000 1110 0011 1000 1101 0111`  
`exponentResult = 0000 0001`
  - Bit 23 of the mantissa of the result is set, so reset it:  
`mantissaResult = 0000 0000 0000 1110 0011 1000 1101 0111`
  - `signResult = 1`
  - `result = 1000 0000 1000 1110 0011 1000 1101 0111`
-