# Computer architectures

| | |
|---|---|
| **Iniziato** | mercoledì, 2 dicembre 2020, 13:54 |
| **Stato** | Completato |
| **Terminato** | mercoledì, 2 dicembre 2020, 13:55 |
| **Tempo impiegato** | 44 secondi |
| **Valutazione** | **0,00** su un massimo di 32,00 (**0**%) |

**Domanda 1**

Risposta non data

Punteggio max.: 3,00

The Tomasulo architecture for superscalar processors with dynamic scheduling and speculation uses reservation stations.

You are requested to

- Explain what reservation stations are and where they are placed in the Tomasulo architecture, listing the modules they are connected to
- Describe the hardware structure of a reservation station
- Summarize when data/information are written/updated in a reservation station
- Explain the advantages stemming from the introduction of the reservation stations in the Tomasulo architecture.

---

**Domanda 2**

Risposta non data

Punteggio max.: 4,00

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: 4 clock periods (pipelined)
- FP divider unit: 6 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 value of the constant k is written in f10 before the beginning of the code fragment.

```
; ******************** MIPS64 ***********************
;  for (i = 0; i < 10; i++) {
;       v5[i] = (v1[i]*v2[i]) + (v3[i]*v4[i])/k;
;  }
```

---

**Domanda 3**

Risposta non data

Punteggio max.: 5,00

Given a 4 x 4 matrix of bytes SOURCE  write a 8086 assembly program which rotates the columns of SOURCE from right to left by 0<=n<=4 positions, with n given by the user. The choice is yours about how to store the matrix in the memory. Please add significant comments to the code and instructions.

Example:

Initial matrix SOURCE
A B C D
E F G H
I J K L
M N O P

if n=1 SOURCE becomes

B C D A
F G H E
J K L I
N O P M

on the other hand, if n=2 SOURCE becomes

C D A B
G H E F
K L I J
O P M N

---

/

**Informazione**

Click here to open the web page of the manual:

https://developer.arm.com/documentation/ddi0337/e/introduction/instruction-set-summary?lang=en

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

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

In all subroutines, you can assume that the matrix is big enough to contains all values.

**Explanation of programming exercises**

Let p(x) be a *n*-th degree polynomial: $p(x) = k_n x^n + k_{n-1} x^{n-1} + k_{n-2} x^{n-2} + \ldots + k_2 x^2 + k_1 x + k_0$

The coefficients $k_i$ are unknown, but we know the values of p(1), p(2), …, p(*n*+1). We want to compute the value of p(*m*), with *m* > *n*, using the method of the divided differences. We use a matrix *M* with *m* rows and *n*+1 column. The element at row *i* and column *j* is indicated as *M*[*i*][*j*]. We apply the following algorithm:

phase 1) the elements on the first column *M*[0][0], *M*[1][0], …, *M*[*n*][0] are initialized with the values of p(1), p(2), …, p(*n*+1)

phase 2) the value of an element *M*[*i*][1] on the second column is set as: *M*[*i*][1] = *M*[*i* +1][0] - *M*[*i*][0]. In this way, the first *n* elements on the second column are set.
The value of the first *n*-1 elements on the third column are computed as the difference of the elements of the second column: *M*[*i*][2] = *M*[*i* +1][1] - *M*[*i*][1]
By applying the same rule to all columns, finally we compute the value of the first element of the last column *M*[0][*n*]

phase 3) the value of *M*[0][*n*] is copied into the first *n*+1 elements of the last column. Then, any element M[*i*][*j*], with *j* < *n* - 1, is computed as: *M*[*i*][*j*] = *M*[*i* - 1][*j*] + *M*[*i* - 1][*j* + 1]

Example: p(x) is a fourth order polynomial, with p(1) = 4, p(2) = 30, p(3) = 120, p(4) = 340, p(5) = 780. We want to compute the value of p(9). We use a matrix with 9 rows and 5 columns.

| | | | | | |
|---|---|---|---|---|---|
| 4 | 26 | 64 | 66 | 24 | LEGEND |
| 30 | 90 | 130 | 90 | 24 | |
| 120 | 220 | 220 | 114 | 24 | phase 1 |
| 340 | 440 | 334 | 138 | 24 | |
| 780 | 774 | 472 | 162 | 24 | phase 2 |
| 1554 | 1246 | 634 | 186 | 0 | |
| 2800 | 1880 | 820 | 0 | 0 | phase 3 |
| 4680 | 2700 | 0 | 0 | 0 | |
| 7380 | 0 | 0 | 0 | 0 | not used |

At the end, we obtain p(9) = 7380. Note: unused elements must be left to zero.

---

/

**Domanda 4**

Risposta non data

Punteggio max.: 4,00

Write a `initializeMatrix` subroutine that receives in input:

1. address of a zeroed block of memory: it represents the matrix
2. address of an array: it contains the values p(1), p(2), …, p(*n*+1)
3. *n*+1: grade of the polynomial + 1, i.e., number of values in the array

The subroutine implements the phase 1 of the algorithm of divided differences. It does not return any value.

Example of calling code

```
         AREA matrixDeclaration, DATA, READWRITE
matrix    SPACE 2000
         AREA arrayInitialization, DATA, READONLY
array     DCD 4, 30, 120, 340, 780


           AREA    |.text|, CODE, READONLY
Reset_Handler  PROC
           LDR r0, =matrix
           LDR r1, =array
           MOV r2, #5    ; number of values in the array
                         ; the grade of the polynomial is r2 - 1
         BL initializeMatrix


         [...]
stop      B stop
           ENDP
```

In the example, the block of memory `matrix` after the subroutine call is:

| | | | | |
|---|---|---|---|---|
| 4 | 0 | 0 | 0 | 0 |
| 30 | 0 | 0 | 0 | 0 |
| 120 | 0 | 0 | 0 | 0 |
| 340 | 0 | 0 | 0 | 0 |
| 780 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |

LEGEND

phase 1

phase 2

phase 3

not used

/

**Domanda 5**

Risposta non data

Punteggio max.: 8,00

Write a `computeDifferences` subroutine that receives in input:

1. address of the matrix (with values set by the `initializeMatrix` subroutine)
2. *n*+1: grade of the polynomial + 1

The subroutine implements the phase 2 of the algorithm of divided differences. It does not return any value.

Optional: the subroutine checks the overflow when computing the differences.

If the result of the subtraction is positive but it is too large to fit in 32 bits, then it is replaced with the greatest positive value that you can store in 32 bits.

If the result of the subtraction is negative but it is too small to fit in 32 bits, then it is replaced with the smallest negative value that you can store in 32 bits.

Max score without the overflow check: 4 points

Max score with the overflow check: 5 points

Example of calling code

```
            AREA    |.text|, CODE, READONLY
Reset_Handler  PROC
            [...]
            LDR r0, =matrix
            MOV r1, #5
          BL computeDifferences
          [...]
stop       B stop
            ENDP
```

In the example, the block of memory `matrix` after the subroutine call is:

| | | | | |
|---|---|---|---|---|
| 4 | 26 | 64 | 66 | 24 |
| 30 | 90 | 130 | 90 | 0 |
| 120 | 220 | 220 | 0 | 0 |
| 340 | 440 | 0 | 0 | 0 |
| 780 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |

LEGEND

phase 1

phase 2

phase 3

not used

/

**Domanda 6**

Risposta non data

Punteggio max.: 8,00

Write a `getPolynomialValue` subroutine that receives in input:

1. address of the matrix (with values set by the `computeDifferences` subroutine)
2. $n$+1: grade of the polynomial + 1
3. $m$

The subroutine implements the phase 3 of the algorithm of divided differences. It returns the value of p($m$).

Optional: the subroutine checks the overflow when computing the sums.

If the result of the addition is positive but it is too large to fit in 32 bits, then it is replaced with the greatest positive value that you can store in 32 bits.

If the result of the addition is negative but it is too small to fit in 32 bits, then it is replaced with the smallest negative value that you can store in 32 bits.

Max score without the overflow check: 4 points

Max score with the overflow check: 5 points

Example of calling code

```
            AREA    |.text|, CODE, READONLY
Reset_Handler  PROC
            [...]
            LDR r0, =matrix
            MOV r1, #5
          MOV r2, #10    ; return value will be p(r2)
          BL getPolynomialValue
stop       B stop
            ENDP
```

In the example, the block of memory `matrix` after the subroutine call is:

| | | | | | |
|---|---|---|---|---|---|
| 4 | 26 | 64 | 66 | 24 | LEGEND |
| 30 | 90 | 130 | 90 | 24 | |
| 120 | 220 | 220 | 114 | 24 | phase 1 |
| 340 | 440 | 334 | 138 | 24 | |
| 780 | 774 | 472 | 162 | 24 | phase 2 |
| 1554 | 1246 | 634 | 186 | 0 | |
| 2800 | 1880 | 820 | 0 | 0 | phase 3 |
| 4680 | 2700 | 0 | 0 | 0 | |
| 7380 | 0 | 0 | 0 | 0 | not used |

**Domanda 7**

Risposta non data

Non valutata

if you had any problems or need more space please use this one here. If not, please this space blank.