Processor Architecture:

Course Project

Francisco Martinez, Sergi Alcaide and Jordi Cardona
PA - MIRI - HPC
Facultat d'Informàtica de Barcelona

January 25, 2017

Contents

List of Figures

Processor brief explanation

In this project we have build a MIPS pipelined and multicycle processor with 2 pipelines paths (the first one executes the R-type, M-type and B-type instructions and the second one executes the multiplication instruction). We have implemented this processor from scratch and the decisions that we have taken we have sought to be the most similar as possible to the MIPS reference standard manual (instructions encoding, signals,...).

The first pipeline is composed by the following stages: Fetch(F), Decode(D), Execute(E) and Write Back (WB).

The second pipeline is composed by the following stages: Fetch(F), Decode(D), Multiply 1 to Multiply 5 (M1, M2, M3, M4, M5) and Write Back (WB). The processor build contains the following features [1]:

Processor particularities:

The processor designed in this project includes the branch checking condition in the Decoder stage not in the ALU stage as usual in order to save one cycle in case of jumping (the instruction fetched that continues the execution has to be removed from the pipeline). So, we can say that our processor includes a worst branch penalty than processors that has a branch predictor engine in the fetch stage but a better penalty that the ones that has the prediction checking in the Execution stage.
In order to make this architectural improvement efficient, we have had to implement a full set of bypasses from Execution(ALU)→Decode and also from MEM(Data Cache)→Decode to receive as soon as possible the result of the register computed in these stages to the checking part in the decode. In addition, we have had to include an adder and a comparator in the Decode stage so as to evaluate if we have to branch (branch not equal or equal) and also to compute the branch address where we have to jump.

Bypasses:

The processor implemented in this project includes all set of bypasses in the fist path pipeline (ALU→ALU, Mem→ALU, Mem→Mem) but not the bypasses between the first and second pipeline. Furthermore, we also have the additional bypasses implemented from all stages of the first pipeline path to the Decoder mentioned in the previous section to improve the branch penalty as we do not have a branch predictor.
So as to make our processor work, we have also implemented the logic bound to these bypasses taking into account the stalls due to instructions and data misses, structural risks (Write Back stages of different pipelines path length) and data dependencies (WAW and RAW).

Cache and memory managing:

This processor also has cache memory inclusion (icache for instructions and dcache for data) that follows a write-back policy that updates the cache values to the memory when a cache line is replaced with dirty bit 1. Both caches are direct mapped caches that have 4 lines of 128 bits. Even though we do not manage unaligned accesses to memory, we manage the arbitration (Arbiter.v) between caches when we have a icache and dcache petition at the same time or when we have a data petition when a icache miss is in progress or inside out. As a principal memory we have used a 3KByte memory with lines also of 128 bits in order to establish easy mapping between caches and memory.

[1] You can also consult all the hazard and bypasses taken into account in this project attached in a excel file

Instruction Set

Instruction/ Syntax	Encoding	Description
Add $rd,$rt,$rs	0000 00ss ssst tttt dddd d000 0010 0000	Adds two registers and stores the result in a register
sub $rd,$rt,$rs	0000 00ss ssst tttt dddd d000 0010 0010	Subtracts two registers and stores the result in a register
sll $d, $s, h	0000 00ss ssst tttt dddd dhhh hh00 0000	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in
mult $d,$s, $t	0000 00ss ssst tttt 0000 0000 0001 1000	Multiplies $s by $t and stores the result in $d
lb $t, offset($s)	1000 00ss ssst tttt iiii iiii iiii iiii	A byte is loaded into a register from the specified address
lw $t, offset($s)	1000 11ss ssst tttt iiii iiii iiii iiii	A word is loaded into a register from the specified address
sb $t, offset($s)	1010 00ss ssst tttt iiii iiii iiii iiii	The least significant byte of $t is stored at the specified address
sw $t, offset($s)	1010 11ss ssst tttt iiii iiii iiii iiii	The contents of $t is stored at the specified address
addi $t, $s, imm	0010 00ss ssst tttt iiii iiii iiii iiii	Adds a register and a sign-extended immediate value and stores the result in a register.
addiu $t, $s, imm	0010 01ss ssst tttt iiii iiii iiii iiii	Adds a register and a sign-extended immediate value and stores the result in a register.(Works also as pseudoinstrucion LI)
beq $s, $t, offset	0001 00ss ssst tttt iiii iiii iiii iiii	Branches if the two registers are equal
ori $t, $s, imm	0011 01ss ssst tttt iiii iiii iiii iiii	Bitwise ors a register and an immediate value and stores the result in a register
lui $t, imm	0011 11– —t tttt iiii iiii iiii iiii	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes

Instructions Behavior

Load (LW)

Fetch: Instruction is fetched from the Instruction cache using the address contained in the program counter register. After a reset of the processor this instruction is loading from 0x1000 address. In case of icache miss, we will inject a bubble to the decode stage and stall the current address of the PC.

Decode: Here the instruction is decoded, as is a Load, we get the register (bits 25:21) from to calculate the address of the data, the register 2(bits 20:16), where we will write the data and finally the signed immediate value (bits 15:0) which we will need to extend the sign up to 32 bits. Control logic is present in order to check that all the sources are correctly updated.

Execute: As explained before in the Execute stage we have the multiplexer for the operands of the ALU. In the case of the Load we will select the data correctly.
3 different cases can appear:

- 00: Default case, get the value directly from the Decode stage
- 01: We get the value from the previous instruction : ALU-ALU bypass
- 10: We get the value from a previous load: MEM-ALU bypass

In the case of the second operand of the ALU, as it is an intermediate we cannot have any dependency, but we must select between the value coming from the reg2 bypass or the immediate. A obvious we will select the second. We will simply add these two operands to obtain the address.

Memory: At this stage we will bring two operands, the destination register and the ALU result (address). We will request the value of the address to the Dcache. In case of Dcache miss we will stall all the previous pipeline (F,D,Exec) and inject bubbles to the WB stage until the Dcache hits. Now we must select the data that comes from the Dcache instead of the data that comes directly from Exec stage.

WB: Now we must select the data coming from the M stage instead of the one coming from the bottom pipeline. We will have also to select the destination register and the write permission that we have been bringing during all the pipeline. Finally we can now write the values to the RegisterFile.

Store (SW)

Fetch: The same behaviour than the Load.

Decode: In this case we select with the same bits the register 1 and register 2, but in this case we must set the destiation register to 0 and the write permission for RegisterFile to 0 and write permission for Mem to 1. Register 2 will be sent as we will need it to write to MEM.

Execute: Here we will find the same multiplexer than the Load, but we also need to take care of the bypasses to register 2. First of all the ALU will add the result of the muliplexer for register 1 and the immediate, which will be the data where to store the value of the multiplexer of register 2.
The muliplexer of register 2 is almost the same than for the register 1:

- 00: Default case, get the value directly from the Decode stage
- 01: We get the value from the previous instruction : ALU-ALU bypass
- 10: We get the value from a previous load: MEM-ALU bypass
- 11: In the case that previous instruction is a Load that writes into our register 2 we must use this bypass

Memory: We will use the Write Permission of Memory to write into Dcache the register 2 (bypassed or not).

WB: Store does not do anything in this stage.

Branch (BNE)

Fetch: The Branch not equal is fetched as the previous explained instructions.

Decode: In this case, the decoder selects the source registers 1 and 2 and the immediate that will be the offset to the address where we want to branch. After that, we ask for these values in register file, provide the results using bypasses (for example ALU→Decode(branch)) or stall until we got the values and then we check if src1 and src2 are not equal. In parallel we also calculate the possible next pc address in case of branch. If they are not equal, the z signal is activated and at the same time the branch destination is provided to the pc. If z signal (in our case called *is_branch*) is activated, we have to remove the fetched instruction (only correct if we do not jump) and inject a bubble to the next decode stage and fetch the new correct branch address calculated in the decoder stage of the BNE instruction.

Execution: This instruction does not have nothing to do in the execution stage (all the calculus and checks have been already done in the previous stage).

Memory: This instruction does not access to memory.

Write Back: This instruction does not write any value in the register file at the end of the pipeline.

Multiply (MUL)

Fetch: The same behaviour as in the previous instructions.

Decode: In this instruction, the decoder picks up the src1, src2 and destination register and takes the values from the decoder (as in the R-Type instructions).

M stages: In this case, instead of using the "normal pipeline path" we will use the alternative path M1, M2, M3, M4, M5, WB. The two data values for the src1 and src2 registers are provided also with the id of the destination value and the wb_write permission that it will be used in the WB stage. In the implemented processor the multiplication is really performed in the M1 stage and the result of this multiplication goes down to the pipeline as long as the cycles go by. For this reason, we do not pass the src1 and src2 values down the M2 to M5 stages because we directly use the multiplication result.

Write Back: Now we must select the data coming from the "bottom pipeline" stage instead of the one coming from the bottom the M stage. We will have also to select the destination register and the write permission that we have been bringing during all the pipeline. Finally, we can now write the value to the RegisterFile.

Performance Tests

We executed the following benchmarks in order to measure the number of cycles our processor needs to execute these typical codes, the clock used is 100ps.

Buffer Sum:

```
.data

vector: .space 512      # Elements*4

.text

main:
    la $t0, vector
    li $t1, 0           # Counter
    li $t2, 128         # Elements

# Let's initialize
loop:
    sw $t1, 0($t0)
    addi $t0, $t0, 4
    addi $t1, $t1, 1
    bne $t1, $t2, loop
    la $t0, vector
    li $t1, 0           # Counter
    li $a0, 0           # Sum

#Summatory
sum:
    lw $t3, 0($t0)
    add $a0, $a0, $t3
    addi $t0, $t0, 4
    addi $t1, $t1, 1
    bne $t1, $t2, sum
    li $v0, 1           # Show result and finish
    syscall
    li $v0, 11
    li $a0, 10
    syscall
    li $v0, 10
    syscall
```

The result of the benchmark is that in register $4 is stored the number 8128.
Time to complete the benchmark: 362500ps
Cycles to complete the benchmark: 3625 cycles

Mem Copy:

Executing this benchmark we have figured out that as we are coping an array from one memory position to another (load followed to store) in addresses that are mapped in the same cache line, we have in every loop iteration 2 data misses (LD miss evicts previous modified line and brings the line, ST miss evicts a non modified line brings the new line). The timing results provided below are using padding in the C matrix (A and B replace the same line but not C).

```
.data
#N_ELEMENTS = 128
#T_ELEMENT = 4
#.align 2
vector_a: .space 512 #N_ELEMENTS*T_ELEMENT
vector_b: .space 512 #N_ELEMENTS*T_ELEMENT
```

```
7
8  .text
9  main:    la $t0, vector_a
10          li $t1, 0 #Counter
11          li $t2, 128 #N_ELEMENTS
12          li $t3, 5
13
14 #Let's initialize a
15 loop:
16          sw $t3, 0($t0)
17          addi $t0, $t0, 4
18          addi $t1, $t1, 1
19          bne $t1, $t2, loop
20
21          la $t0, vector_a
22          la $t1, vector_b
23          li $t2, 0 #Counter
24          li $t3, 128 #N_ELEMENTS
25
26 #Let's initialize b
27 bloop:
28          lw $t4, 0($t0)
29          sw $t4, 0($t1)
30          addi $t0, $t0, 4
31          addi $t1, $t1, 4
32          addi $t2, $t2, 1
33          bne $t2, $t3, bloop
34
35          li $v0, 10
36          syscall
```

The result of the benchmark is that we created the vector b in memory, except the last line which is in cache, full of 5's.
Time to complete the benchmark: 282100ps
Cycles to complete the benchmark: 2821 cycles

Matrix Multiply:

```
1  .data
2  #ROWS = 5
3  #COLUMNS = 5
4  #T_ELEMENT = 4
5
6  matrix_a: .space 100
7  matrix_b: .space 100
8  matrix_c: .space 100 # 5 * 5 * 4
9
10 .text
11 main:    la $t0, matrix_a
12          la $t1, matrix_b
13          la $t2, matrix_c
14          li $t3, 16 #ROWS
15          li $t4, 16 #COLUMNS
16
17          li $t5, 0 # i = 0
18
19 loop_i:
20          li $t6, 0 # j = 0
21
22 loop_j:
23          li $t7, 0 # k = 0
24          li $s0, 0 # c[i][j]
25
```

```
26 loop_k:
27         # Let's get a[i][k]
28         mul $t8, $t5, $t4 # ROW * NCOLUMNS
29         add $t8, $t8, $t7 # (ROW * NCOLUMNS) +COLUMN
30         sll $t8, $t8, 2 # ((ROW * NCOLUMNS) +COLUMN)* T_ELEMENT
31         add $t8, $t0, $t8 # @ + ((ROW * NCOLUMNS) +COLUMN)* T_ELEMENT
32         lw $s1, 0($t8) # a[i][k]
33
34         # Let's get b[k][j]
35         mul $t8, $t7, $t4  # ROW * NCOLUMNS
36         add $t8, $t8, $t6 # (ROW * NCOLUMNS) + COLUMN
37         sll $t8, $t8, 2 # ((ROW * NCOLUMNS) + COLUMN)* T_ELEMENT
38         add $t8, $t1, $t8 # @ + ((ROW * NCOLUMNS) +COLUMN)* T_ELEMENT
39         lw $s2, 0($t8) # b[k][j]
40
41         mul $s1, $s1, $s2 # a[i][k] * b[k][j]
42         add $s0, $s0, $s1 # c[i][j] = c[i][j] + a[i][k] * b[k][j]
43
44         addi $t7, $t7, 1 # k++
45         bne $t7, $t4, loop_k
46
47 done_k: mul $t8, $t5, $t4 # ROW * NCOLUMNS
48         add $t8, $t8, $t6 # (ROW * NCOLUMNS)  +COLUMN
49         sll $t8, $t8, 2 # ((ROW * NCOLUMNS)  +COLUMN)* T_ELEMENT
50         add $t8, $t2, $t8 # @ + ((ROW * NCOLUMNS) +COLUMN)* T_ELEMENT
51
52         sw $s0, 0($t8) # c[i][j] = c[i][j] + a[i][k] * b[k][j]
53
54         addi $t6, $t6, 1 # j++
55         beq $t6, $t4, loop_j
56
57 done_j: addi $t5, $t5, 1 # i++
58         bne $t5, $t3, loop_i
59
60 done_i: li $v0, 10
61         syscall
```

Explanation and Results:

Figure 1: Matrix multiply dataset $A * B = C$

(a)

Input matrix A				
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

(b)

Input matrix B				
1	3	2	6	4
4	8	5	3	2
5	6	9	80	4
1	2	3	5	8
8	9	4	6	3

(c)

Input matrix C				
68	90	71	86	67
163	230	186	226	172
258	370	301	366	277
353	510	416	506	382
448	650	531	646	487

Time to complete the benchmark: 1049800ps
Cycles to complete the benchmark: 10498 cycles

Future Work:

Because of the limited time We still have things to implement in our processor:

- Store Buffer (not included in the processor project even is implemented).
- Reorder Buffer (ROB) or History File (HF)
- Precise exceptions
- Virtual Memory and syscalls