PA - Project Report

Jaume Bosch, Marcos Maroñas

April 23, 2018

1 Pipeline stages and block diagram

The stages that our processor has are:

- **iTLB**. This stage translates the virtual memory address of the instruction (PC) into the physical one. On one hand, if the processor is in system mode, the iTLB is disabled because the processor works with physical addresses. On the other hand, if the iTLB does not contain a valid translation for the required address an exception will be generated when the instruction reaches the decode stage (before the instruction does not have a valid ROB Idx to be inserted).
- **Fetch**. This stage retrieves the data contained in the physical memory address (PC). If the cache does not contain the data, this stage stalls the previous stages until the cache receives the data from the memory interface.
- **Decode**. This stage decodes the instruction stored in memory: it assigns a ROB Idx to the instruction if needed, looks for the newest values of used registers and prepares the input values to the ALU or Multiplier. Moreover, if the Hazards logic detects a conflict the stage may stall the previous stages until the instruction can be issued.
- **ALU**. This stage applies an arithmetic operation over the input data according to the instruction. If the instruction is an arithmetic operation it can be directly inserted into the ROB without use any other stage.
- **Multiplier**. This is the combination of the five stages that a multiplication takes in our processor, it takes two integers and calculates the multiplication of them. The first four stages just flop the input data and the product is actually done in the fifth stage. The last stage is the responsible to insert the instruction to the ROB.
- **dTLB**. This stage translates the virtual memory address of the Loads and Stores into the physical one. On the other hand, if the dTLB does not contain a valid translation for the required address an exception is generated and pushed to the ROB. On the other hand, if the instruction is an Store it is inserted to the ROB from this stage because our processor provides precise exceptions and the write is done from the ROB.
- **dCache**. This stage retrieves the data contained in the physical memory address calculated in the Loads and/or writes some data into a memory position. One one hand, if the cache does not contain the required data, this stage stalls the previous stages until the cache receives the data from the memory interface. On the other hand, all writes take one cycle because the cache contain a store buffer that holds them (see Figure 2).
- **WB**. This stage writes the result of an instruction into the encoded register. This stage is executed when the instruction is the oldest one stored in the ROB.

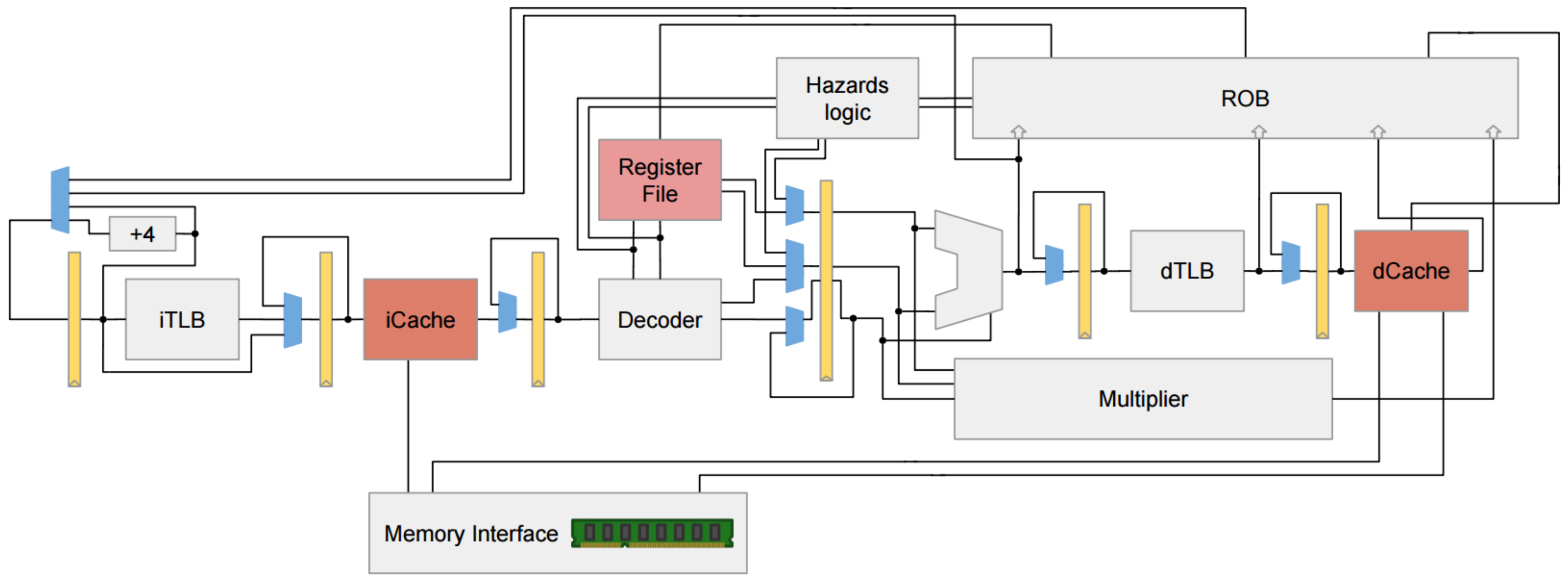

Figure 1: Full pipeline diagram

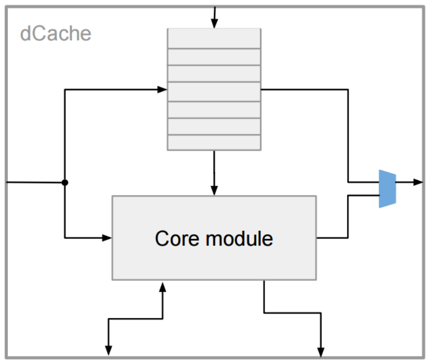

Figure 2: Data cache diagram

2 General information

The register file of our processor contains 4 additional special registers. They are `rm0`, `rm1`, `rm2` and `rm4`. Respectively, they are used to save the faulting PC when an exception raises, save the faulting address when it is a TLB exception, save additional information regarding the exception and hold the current privilege of the machine.

Our processor boots in OS_MODE. This means that the special register `rm4` contains a 1. The processor boots from address 0x1000. Given that it starts in OS_MODE, the virtual memory is disabled so this is a physical address. The instruction in address 0x1000 is an `iret` which jumps to the user code. The same instruction replaces the 0 in `rm4` with a 1, which means USER_MODE. Hence, virtual memory is enabled. The user code starts at virtual address 0x1000 (physical address 0x9000).

In physical address 0x2000 there is the exception handler. When the processor raises an exception, it goes to OS_MODE and jumps to the mentioned address. At the end of the handler, an `iret` instruction turns back into USER_MODE and returns to the faulting PC.

The `hazards logic` module is responsible to check if the instruction in the *decode* can be forwarded to the next stage, or it has some data dependence that is not yet accomplished. Moreover, the module gathers information from the *ALU*, *dCache*, *Multiplier* and *ROB* in order to bypass the value that will be wrote into the registers in the future (avoiding the stall).

The store buffer contained in the data cache (*dCache*) holds in only one cycle the cache writes and keeps the information until the real cache module processes it. One cache write requires only one cycle if it is a hit, and around eight cycles if it is a miss (there may be conflicts with an *iCache* miss). When one instruction tries to read a memory position, we have to check both places: cache core module and store buffer. If the store buffer contains some write to the address, this will be the value read by the instruction. When the store buffer holds byte-writes and we are trying to read a whole word, we stall the pipeline until the write is effectively done in the corresponding cache line to simplify the implementation.

3 Add/Sub

An add/sub instruction uses the following stages: iTLB, Fetch, Decode, ALU, WB. This instruction can stall in the Fetch stage due to a instruction cache miss and in the Decode stage due to data hazard. Between the ALU and the WB, the instruction goes to the ROB which commits the instruction, effectively writing the result in the respective register, when it is the oldest instruction in the ROB. Moreover, if there is a instruction TLB miss, the exception is raised also by the ROB.

4 Mul

A mul (multiplication) instruction uses the following stages: iTLB, Fetch, Decode, MUL (M0, M1, M2, M3, M4), WB. This instruction can stall in the Fetch stage due to a instruction cache miss and in the Decode stage due to data hazard. Between the M4 and the WB, the instruction goes to the ROB which commits the instruction, effectively writing the result in the respective register, when it is the oldest instruction in the ROB. Moreover, if there is a instruction TLB miss, the exception is raised also by the ROB.

5 Load

A load instruction uses the following stages: iTLB, Fetch, Decode, ALU, dTLB, dCache, WB. This instruction can stall in the Fetch stage due to a instruction cache miss and in the Decode stage due to data hazard. In the ALU, the instruction calculates the address to be read by adding the offset (immediate) and the base address (value hold in one register). Between the dCache and the WB, the instruction goes to the ROB which commits the instruction, effectively writing the result in the respective register, when it is the oldest instruction in the ROB. Moreover, if there is a instruction TLB miss or a data TLB miss, the exception is raised also by the ROB.

6 Store

A store instruction uses the following stages: iTLB, Fetch, Decode, ALU, dTLB, dCache. This instruction can stall in the Fetch stage due to a instruction cache miss and in the Decode stage due to data hazard. In the ALU, the instruction calculates the address to be written by adding the offset (immediate) and the base address (value hold in one register). Between the dTLB and the dCache, the instruction goes to the ROB which commits the instruction, effectively writing the value in the respective memory address, when it is the oldest instruction in the ROB. Moreover, if there is a instruction TLB miss or a data TLB miss, the exception is raised also by the ROB.

7 Branch

A branch instruction uses the following stages: iTLB, Fetch, Decode, ALU. This instruction can stall in the Fetch stage due to a instruction cache miss and in the Decode stage due to data hazard. When it reaches the ALU stage the processor knows if the branch must be taken and the PC has to be modified (discarding the instructions in the iTLB, Fetch and Decode stages). Moreover, if there is a instruction TLB miss, the exception is raised also by the ROB.

8 Performance Tests

This section provides performance results for three different benchmarks presented following.

8.1 Buffer_sum

The first benchmark is `buffer_sum`. The high-level code is shown in Figure 3. As can be observed in the mentioned figure, this benchmark just sum all the values of the array into a variable.

```c
int a[128], sum = 0;
for(int i = 0; i < 128; i++)
    sum += a[i];
```

Figure 3: High-level code of buffer_sum benchmark.

In order to check the correctness, we have initialized all the values of a to its corresponding position like in Figure 4.

```
for(int i = 0; i < 128; i++)
    a[i] = i;
```

Figure 4: Initialization of buffer_sum benchmark.

Our processor took 16800ns to complete the benchmark. Since each cycle lasts for 10ns, the benchmark needs **1680 cycles** to be completed. Our assembler implementation contains 774 instructions, taking into account those repeated in the loop. Therefore, the IPC of our processor in this benchmark is $\simeq 0.46$.

8.2 Mem_copy

The second benchmark is `mem_copy`. Figure 5 presents the high-level code. In this case, there are two arrays, the first array is initialized to a given number and then its contents are copied to the other array.

```
int a[128], b[128];
for(int i = 0; i < 128; i++)
    a[i] = 5;
for(int i = 0; i < 128; i++)
    b[i] = a[i];
```

Figure 5: High-level code of mem_copy benchmark.

For the completion of the benchmark, our processor took 31650ns, and so, 3165 cycles. The instructions executed to complete the benchmark were 1421. Hence, in this benchmark we have an IPC $\simeq 0.45$.

8.3 Matrix_multiply

The last benchmark is `matrix_multipliy` whose high-level implementation can be observed in Figure 6. This last benchmark performs the multiplication of two matrix and stores the result in a third matrix.

```
int a[128][128], b[128][128], c[128][128];
for(int i = 0; i < 128; i++) {
    for(int j = 0; j < 128; j++) {
        c[i][j] = 0;
        for(int k = 0; k < 128; k++ {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Figure 6: High-level code of matrix_multiply benchmark.

Three matrices of 128x128 means a huge amount of memory, so we restricted the size of our matrices to 16x16. Regarding the initialization, it is presented in Figure 7.

This is the longest execution. This benchmark needs 1887990ns or 188799 cycles to finish. The total number of executed instructions is 63566, so for this benchmark IPC $\simeq 0.3367$.

```
// Consider that matrices are 16x16
for(int i = 0; i < 16; i++) {
    for(int j = 0; j < 16; j++) {
        a[i][j] = i*16 + j;
        b[i][j] = 256 + i*16 + j;
    }
}
```

Figure 7: Initialization of matrix_multiply benchmark.