MIPS simulation software

Pawel Knap
pmk1g20@soton.ac.uk
Student ID: 431573975
22/04/2022

1. Design

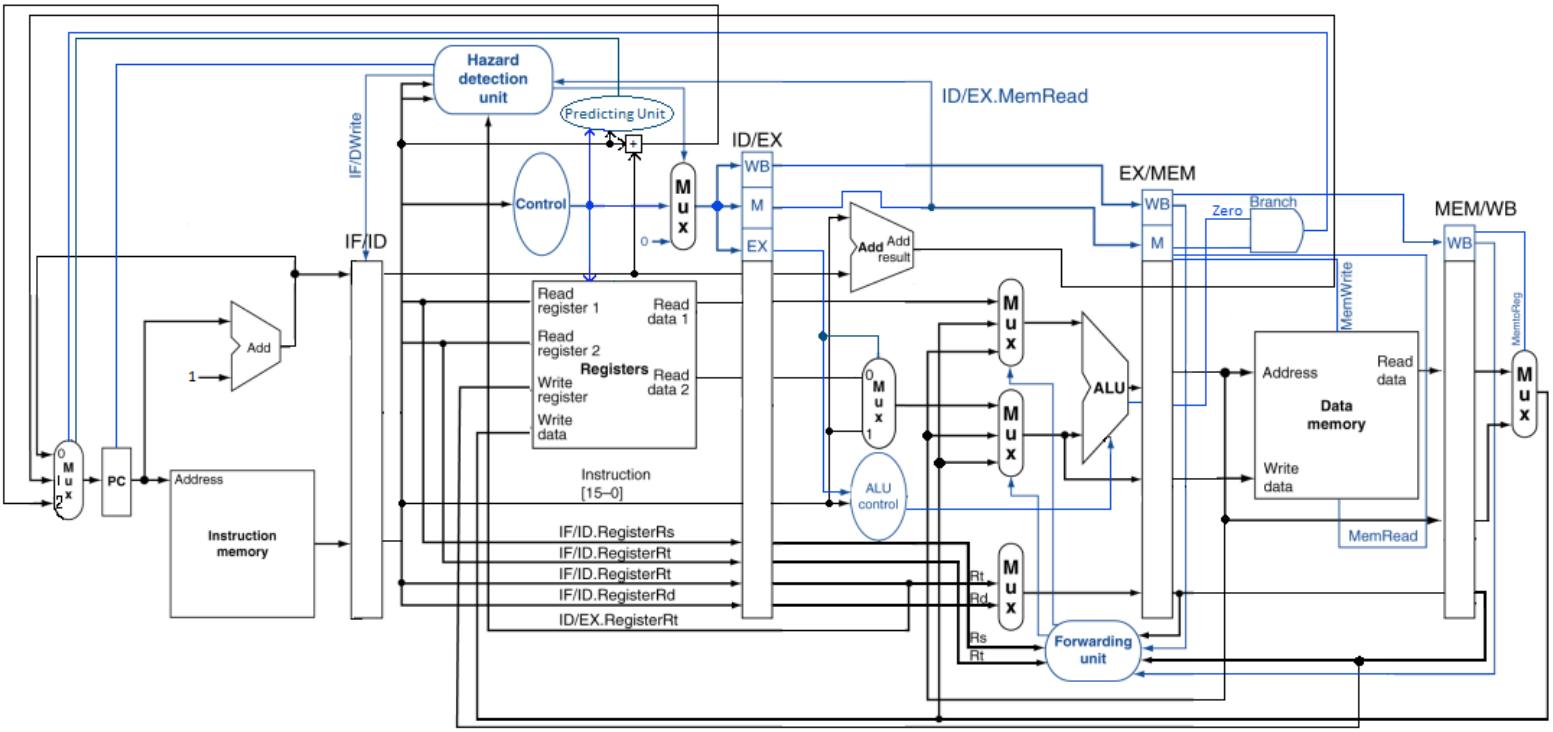

Figure 1. Diagram of implemented “processor” (adapted from the lecture slides)

The implemented processor is a 5-stage pipelined computer following a straightforward fetch/decode/execute/write-back/memory access cycle. It has separate data and instruction memory and allows execution of a few MIPS assembly commands such as immediate and normal addition, AND, OR & XOR operations, subtraction, store/load a word and branch if equal/not equal. Tables I and II show the instruction set.

Table I R-type operation set

Type	Opcode (6)	Register addr1 (5)	Register addr2 (5)	Aim register addr (5)	(11 bits)
Addition	000000	XXXXX	XXXXX	XXXXX	00000 100000
Subtraction	000000	XXXXX	XXXXX	XXXXX	00000 100010
AND	000000	XXXXX	XXXXX	XXXXX	00000 100100
OR	000000	XXXXX	XXXXX	XXXXX	00000 100101
XOR	000000	XXXXX	XXXXX	XXXXX	00000 100110

Table II I-type operation set

Type	Opcode (6 bits)	(5 bits)	(5 bits)	(16 bits)
Immediate addition	001000	Register addr1	Aim register addr	Value
Immediate AND	001100	Register addr1	Aim register addr	Value
Immediate OR	001101	Register addr1	Aim register addr	Value
Immediate XOR	001110	Register addr1	Aim register addr	Value
Store a word	101011	Rs	Register addr1	Offset
Load a word	100011	Rs	Register addr1	Offset
Branch if equal	000100	Register addr1	Register addr2	Offset
Branch if not equal	000101	Register addr1	Register addr2	Offset

Table I should be self-explanatory, but more ambiguity may arise when using table II commands, especially load, store and branch instructions. The Rs value for store/load operations determines the register number. The content of this register is then added to the offset. The result is a data memory address, to which the content of the specified register (Register addr1) will be saved for store command. Whereas, the load instruction moves the

content of the calculated memory address to the specified register (Register addr1). When it comes to the branch instruction, the contents of two specified registers are compared together – using subtraction and the ZERO flag of ALU. If the result of this operation makes the processor branch, then the PC (Program Counter) is changed. The PC takes the value of the sum of itself and offset (two's complementary number). That happens because the offset is relative to the place where the program is. In other words, the jump is relative to the jump command occurrence.

Both registers and data memory are implemented in almost the same way. The number of registers and memory addresses can be defined by the user, but the size of a word is 32 bits. All registers and memory values are set to 0 at the class instantiation. Registers and data memory are programmed as an RF & DM class respectively. Both classes have read, write and output functions. As the names suggest, the first two functions are responsible for reading out and writing in the data. Whereas the last one creates a text file with the state of the memory/registers. In this file every line represents another address, starting from 0, and the value shown is an integer value that is present under this address. An example of such a file is shown in figure 10 in the appendix. The code for both RF & DM classes was based on the Mandar Mhaske project[1].

```
class RF{ //this class simulates registers
    public:
        int Reg_data;
      RF(){
        Registers.resize(32); //32 registers
        Registers[0] = (0);} //set all register to 0

        int readRF(int Reg_addr) {   //reading the value from register
            Reg_data = Registers[Reg_addr];
            return Reg_data;}

        void writeRF(int Reg_addr, int Wrt_reg_data){//writting the value to register
            Registers[Reg_addr] = Wrt_reg_data;}

    void outputRF(int n){//saving the state of registers to text file
      ofstream rfout;
      rfout.open("RFresult.txt",std::ios_base::app);
      if (rfout.is_open()){
        rfout<<"State of Registers at cycle " << n << ":\t" <<endl;
        for (int j = 0; j<32; j++)
          rfout << Registers[j]<<endl;}
      else cout<<"Unable to open file";
      rfout.close();}
    private:
        vector<int>Registers;
};
```

Figure 2. Class simulating registers

```
class DM{    //this class simulates data memory
    public:
        int data_MEM;
      DM(){ //500 memory addresses
        Registers.resize(500);
        Registers[0] = (0);} //set all to 0

        int readDM(int addr){//reading a value from memory
            data_MEM = Registers[addr];
            return data_MEM;}

        void writeDM(int addr, int Wrt_data){//writing a value to memory
            Registers[addr] = Wrt_data;}

    void outputDM(int n){//saving the state of the memory to text file
      ofstream rfout;
      rfout.open("DMresult.txt",std::ios_base::app);
      if (rfout.is_open()){
        rfout<<"State of Data Memory at cycle " << n << ":\t"<<endl;
        for (int j = 0; j<201; j++)
          rfout << Registers[j]<<endl;}
      else cout<<"Unable to open file";
      rfout.close();}
    private:
        vector<int>Registers;
};
```

Figure 3. Class simulating data memory

2. Functionality

2.1 Classes

As mentioned above, there is a special class for registers and data memory. What is more, there is a class for control. It consists of all control signals and every block of processor (IF, EX, MEM, WB) has its own instance of this class. At the end of the cycle, the signals from one stage are forwarded to another by simply equating control class instantiations.

The last class in my program is the so-called InstMemDec which is a shortcut for Instruction Memory Decoding. This class includes a function that breaks the program instruction into parts and stores them. These parts are the signals going out of the IF/ID register e.g. register addresses, opcode or an immediate value. For these signals that go to the execute block, the same technique of forwarding as for control signals is used.

2.2 Functions

The program consists of 7 functions. ALU function represents the ALU block of the processor. All mathematic operations are conducted by this function. The type of the executed operation is specified by the ALUcontrol function, which sets the control signal for ALU. This function corresponds to the block with the same name in figure 1. Similarly, the forwadingUnit and HazardDetectionUnit functions match the blocks shown in the processor schematic. They prevent the miscalculations or erroneous operations, and are responsible for forwarding and hazard detection - thus they are responsible for stalling. The Forwarding unit decreases the number of stalls by directly sending required data from MEM or WB stage to the execution block.

When the stall happens the PC is not updated therefore, the current instruction is decoded again (in the decode stage) and the following instruction is fetched again (in the fetch stage). What is more, if there is a stall in EX, MEM & WB stages, all control signals for them are 0 thus there is no operation in these blocks. Stalls usually happen in single blocks, not in the whole processor at once.

Moreover, the control function sets all control signals according to different opcodes. Its arguments are the aforementioned control class instantiation (IF), which has its control signals changed by this function, and the opcode, which decides what signals should be true (high). The read instruction function (ReadIns) uses the current PC value in order to find the correct instruction in the program memory. It is used in the fetch stage of the processor. The last function of my program is BinToDec which can convert two's complement or raw binary numbers to a decimal representation.

2.3 Main function

The first operation in the main function is reading the program from a text file and saving it in the array. The program asks the user to enter the name of the file every time. This feature improves user experience as well as increases the flexibility of the program.

The next step is an instantiation of all variables required for the infinite loop. This endless loop is the most important part of the program. Every repetition represents one clock cycle. By adjusting a few lines of the code, the user can choose the preferred way of code execution inspection. For example, the program execution can be observed cycle by cycle from the beginning or from a certain spot.

The endless loop can be divided into 6 regions. They represent fetch, decode, execute, write-back, memory access stages and registers between these blocks.

2.4 Fetch block

The first action in the fetch block is updating the program counter (PC) value. If-else statements are used to distinguish between three different ways of doing it. The "jump" signal is high when the processor predicts that there will be a jump. The "branch" signal is high when there should be a jump. The prediction is always made two clock cycles earlier than the decision about the branch is calculated, therefore the "branch" signal has to correct mistakes done by "jump". As a result, "branch" is disabled if the prediction was correct, and it is enabled if the prediction was incorrect (first and third row of Table III). Therefore, in the first case, the program does not jump the second time to the same place, and in the second case, it fixes its mistake and jumps back to the same PC value as before the prediction. Therefore, "jump" is high when the processor predicts that there should be a jump, and the "branch" is high only if the prediction was incorrect. When they are true, the PC value is updated with the new address. "Branch" and "jump" have separate branch address calculation algorithms. These two signals exhaustively cover all possible jump behaviours, as shown in Table III. If both "jump" and "branch" signals are false, and there is no stall, the PC value should be incremented by 1 in every cycle.

The first prediction system implemented was based on typical branch behaviour. It predicts the backward branches being taken, and the forward branches being not taken. This rule is applied as the programs typically go

around in backward loops and rarely jump forward. It should be remembered that every misprediction undeniably introduces a loss of cycles due to stalling of the processor blocks.

Table III "Jump" and "Branch" signals

Jump or not?	Prediction	"jump" signal	initial "branch" signal	"branch" signal
Jump	Jump	True	True	False
Jump	No jump	False	True	True
No Jump	Jump	True	False	True
No Jump	No jump	False	False	False

The next command of the fetch block is calling the ReadIns function. It reads a specified (by the PC) instruction from instruction memory and puts it in the integer array "instruction". The fetch block prints information such as the actual value of PC, instruction in binary format and stall warning. When this block is stalled PC value does not change. This situation occurs when the "stall" signal is high.

2.5 Decode block

First of all, the decode function of the InstMemDec class is used in order to split the instruction into parts such as register addresses, opcode or an immediate value. In this block, the HazardDetectionUnit and control functions are called to set all control signals and stall signals. Moreover, the decoded addresses are used to read the values stored in registers. The program displays information about the current state of this block – whether there is a stall or not. What is more, current opcode, register addresses and data read from them are shown on the terminal.

2.6 Execute block

Here the need for forwarding is checked by the use of forwardingUnit function. It sets the forwardA and forwardB signals which then play a crucial role in determining what arguments should be supplied to the ALU. Then the correct control signal is set for ALU by the ALUcontrol function. It determines what type of operation should ALU perform. After updating some arguments, the ALU function is called. The arguments supplied to it are determined by the ALUSrc, forwardA and forwardB signals. The results of the ALU operation are displayed unless the stall occurs. In the end, the PC value for jump/branch is calculated and the register address for saving the ALU result is decided.

2.7 Data memory block

Here the "branch" signal is calculated and then if there is no stall, and depending on the control signals, the data may be saved to or read from the memory. Naturally, the information about this action is displayed. Moreover, it is checked whether taking a jump was a good prediction. A different variation of the "correct" and "branch" signals determines it and sets the control signals and PC value so that the incorrect decision is undone or the correct decision is maintained, as explained in paragraph 2.4.

2.8 Write back block

Then in the write back stage, the mathematic operation result or a variable read from the data memory is saved into the registers. It happens only when there is no stall and the appropriate control signals are on. The information about the saved value and register address is displayed.

2.9 Register Simulation and stall control

The last part of the endless loop in some sense simulates the behaviour of the registers between stages. It is done by moving the data from variables used in the previous stage to the next stage. The order of instructions is crucial as the data should be moved only to the next stage every clock cycle and not ripple through the whole processor at once. Some commands here, in the abrupt case of the stall, set the control signals in order to stop any operation within the certain blocks. Moreover, the jump prediction algorithm can be found in this section.

3. Basic Testing

A simple program testing all implemented instructions is shown in figure 4. Furthermore, this code tests forwarding for immediate operation, when both EX and MEM hazards occur and forwarding for non-immediate operations in the first argument when an EX hazard occurs (so-called forwardA=2). This code displays capabilities of branch predicting for both branch if equal/not equal commands when the jump is backwards. The behaviour when the prediction was both correct and incorrect can be observed here as well. The misprediction of the branch causes 11 stalls in different blocks, which is equal to 3 clock cycles wasted - one more than when the processor would be stalled until the decision about the branching is made. In the case of correct prediction, 1 clock cycle is saved compared to the aforementioned processor waiting.

```
JUMP1:  R1=3; //immediate addition R1=R0+3;
        R5=R1 ^ 6; //immediate XOR R5=5
        R3=R1 & 6; //immediate AND R3=2
        R2=5; //immediate addition R2=R0+5;
        R4= R2 | 9 //immediate OR R4=13
        sw R1,7; //store register R1 in memory address 7
JUMP2:  R12=R12 + 1; //increment R12 by 1
        be R12,R3,JUMP1; //branch if R12=R3
        R6=R1 + R5; //addition R6=8
        R7=R1 - R5; //subtraction R7=-2
        R8=R1 & R5; //AND R8=1
        R9=R1 | R5; //OR R9=7
        R10=R1 ^ R5; //XOR R10=6
        lw R11,7; //R11=3
        bne R12,R5,JUMP2; //branch if R12!=R5
```

Figure 4. First test program description

All mentioned above commands and operations of the program work as expected. The commented out parts of the code helped during testing, indicating the details about what is going on inside the processor. They can be used at the user's discretion and depending on the intended goal.

The code shown in figure 5 tests the forward jumps for both branch if equal/not equal commands. The behaviour of correct as well as incorrect prediction can be observed. No block is stalled if the prediction about forward jump (predicting no jump) is correct. Therefore program terminates faster by two cycles compared to stalling the processor until the decision about jump is made. Unfortunately, incorrect prediction leads to 3 clock cycles wasted, one more than when the processor waits for the branch decision.

The forwarding for the first and second argument when the MEM hazards occur (so-called forwardA=1 and forwardB=1) as well as forwarding to the second ALU argument when EX hazard occurs (forwardB=2) are verified by the below code.

These two test programs verified the correct execution of all implemented MIPS commands and forwarding operations. Additional testing was carried out on the showcase program. Both test programs in binary form, as well as the showcase program in binary, can be seen in the appendix.

```
JUMP3:  R12=R12 + 1; //increment R12 by 1
        R1=3; //immediate addition R1=R0+3;
        R5=R1 ^ 6; //immediate XOR R5=5
        R3=R1 & 6; //immediate AND R3=2
        R2=5; //immediate addition R2=R0+5;
        be R12,R3,JUMP1; //branch if R12=R3
        bne R12,R3,JUMP2; //branch if R12!=R3
        R4= R2 | 9 //immediate OR R4=13
        sw R1,7; //store register R1 in memory address 7
        R6=R1 + R5; //addition R6=8
        R7=R1 - R5; //subtraction R7=-2
JUMP1:  R8=R1 & R5; //AND R8=1
        R9=R1 | R5; //OR R9=7
JUMP2:  R10=R1 ^ R5; //XOR R10=6
        bne R12,R5,JUMP3; //branch if R12!=R5
```

Figure 5. Second test program description

4. Showcase Testing

The showcase program, shown in figure 6 consists of 12 lines. The first five of them set the correct values in the registers. Then there are two loops, in the first one the square of an x number is calculated by adding x value to

itself x times. The outside loop is responsible for saving the result in the correct place within data memory – memory address x for x number. The value of x is also incremented by 1 and the number of repetitions of the internal loop is set.

This program needs 82019 clock cycles to calculate and save squares of all numbers between 0 and 200. For comparison, the same program needs 122215 clock cycles to finish when there is no prediction algorithm and the processor is stalled until the branch decision is made. The amount of clock cycles is much smaller due to the use of the forwarding and jump prediction, but it could be even smaller if the instruction just after the branch would be performed.

```
        R1=0; //a=0; immediate addition R1=R0+0
        R4=1; //(number of loops) b=1; immediate addition R4=R0+1
        R6=1; //immediate addition R6=R0+1
        R7=201; //immediate addition R7=R0+201
JUMP1:  R5=0; //(sum) immediate addition R5=R0+0

JUMP2:  R5=R5+R1; //sum=sum+a
        R4=R4-1; //b=b-1
        bne R4,R0,JUMP2; //if R4!=0 goto JUMP2

        sw R5,R1; //save R5 to memory address R1 //0 address 0,1 address 1, 4 address 2 ...
        R4=R1+1; //set number of loops for next variable
        R1=R1+1; //next variable = current variable + 1
        bne R1,R7,JUMP1; //repeat multiplication operation for numbers between 0 and 200
```

Figure 6 Showcase program description

5. Conclusions

My design delivers excellent means of testing simple MIPS programs because it allows for high observability. The user is able to work out what each stage is doing in each clock cycle and check the content of memory/registers whenever it is needed. My code provides flexibility, as it is up to the user to choose how much he wants to see. Moreover, the program execution can be inspected cycle-by-cycle for easy debugging and learning of processor basics. Observation may start from the beginning or any clock cycle that the user chooses. Messages in the terminal window are neat, tidy, well-structured and easy to understand as can be seen in figure 11 in the appendix. A huge advantage of this program is that its behaviour is identical to the physical MIPS processor. All basic functions of MIPS were implemented and what is more, they were implemented in the software the same way as they are implemented on hardware in a real MIPS processor. The design goal was to create a mean of testing MIPS programs without the chip.

Looking back at this coursework, I must admit that it was a demanding project. I definitely spend more time on it than expected. However, I am happy with the outcomes. By that, I mean both deliverables and acquired knowledge. Now, the MIPS processor operations do not hide any secrets from me. Moreover, I refreshed my C/C++ coding expertise, good programming practices, as well as critical and logical thinking skills. In my opinion, I gained a lot of knowledge and finally understand the processor behaviour thoroughly. It was an enjoyable project (especially programming as I am a better coder than a writer) that encouraged me to learn more about computer architecture.

When it comes to further work, I cannot skip the fact that my design uses machine code as an input. It may become an issue for some users as not everybody has a full MIPS toolchain capable of converting high-level language code to machine instruction, and saving it into a text file. Therefore, it is necessary to add a simple assembly/machine code compiler. Such a compiler should allow for high flexibility when it comes to the structure of the code. Any spaces or comments should not have any influence on the program execution, as it has now. The performance of the code may be improved by using more advanced branch prediction, for example, dynamic branch prediction instead of static one. Moreover, a simple GUI interface allowing for a high degree of flexibility, so that the user can choose what information he wants to see should be created. Furthermore, more MIPS instructions should be added.

If the one command after branch instruction would be executed, the number of clock cycles needed to calculate the squares of all integers between 0 and 200 would decrease. However, such a change would require alternations in the test & showcase codes. The smallest number of clock cycles required for the execution of the showcase program would be achieved by using one of the multiplication algorithms, for example, the Booth algorithm. In such a case, the computer architecture would have to be built just for this algorithm. This is, of course, not the normal way of doing things, as most processors are general-purpose.

6. Appendix

6.1 Basic Testing

```
*******Line must be empty*******
00100000000000010000000000000011
00111000001001010000000000000110
00110000001000110000000000000110
00100000000000010000000000000101
00110100010001000000000000001001
10101100000000010000000000000111
00100001100011000000000000000001
00010001100000111111111111111001
00000000001000100011000000100000
00000000001000100011100000100010
00000000001000100100000000100100
00000000001000100100100000100101
00000000001000100101000000100110
10001100000010110000000000000111
00010101100001011111111111111000
```

Figure 7. First test program in binary

```
*******Line must be empty*******
00100001100011000000000000000001
00100000000000010000000000000011
00111000001001010000000000000110
00110000001000110000000000000110
00100000000000010000000000000101
00010001100000110000000000000110
00010101100000010000000000000111
00110100010001000000000000001001
00110100010001000000000000001001
00000000001000100011000000100000
00000000001000100011100000100010
00000000001000100100000000100100
00000000101010000100100000100101
00000000001010000101000000100110
00110100010001000000000000001001
00010101100001011111111111110001
```

Figure 8. Second test program in binary

6.2 Showcase program testing

```
*******Line must be empty*******
00100000000000010000000000000000
00100000000001000000000000000001
00100000000001100000000000000001
00100000000001110000000011001001
00100000000001010000000000000000
00000000101000010010100000100000
00000000100001100010000000100010
00010100100000001111111111111110
10101100000100101000000000000000
00100000001001000000000000000001
00100000001000010000000000000001
00010100001001111111111111111001
```

Figure 9. Showcase program in binary

```
State of Data Memory at cycle 82020:
0
1
4
9
16
25
36
49
64
81
100
121
144
169
196
225
256
289
324
361
400
441
484
529
576
625
676
729
784
841
900
961
1024
```

Figure 10. State of first 33 addresses of data memory at cycle 82020

```
------------CYCLE 62------------
**FETCH**
PC is 11
Instruction: 00100000001000010000000000000001
**DECODE**
Opcode: 8
Register_1 address: 1
Register_2 address: 4
Read_data_1 3
Read_data_2 0
**EXECUTE**
Immediate operation no forwarding
3 0
ALU_result 3
Zero 0
**MEMORY**
/////STALL/////
**WRITE BACK**
/////STALL/////

------------CYCLE 63------------
**FETCH**
PC is 12
Instruction: 00010100001001111111111111111001
**DECODE**
Opcode: 8
Register_1 address: 1
Register_2 address: 1
Read_data_1 3
Read_data_2 3
**EXECUTE**
Immediate operation no forwarding
3 1
ALU_result 4
Zero 0
**MEMORY**
Saving 9 in address 3
**WRITE BACK**
/////STALL/////
```

Figure 11 Example of the program interface. Here we can observe, saving a square of 3 in data memory address 3

REFERENCES

[1] Mandar Mhaske (MM10435@NYU.EDU) “Lab 1 - Pipelining without forwarding or stalling or BEQ” https://github.com/ninawekunal/MIPS-Simulator