1st Project
Advanced Computer Architectures

João Ramiro and Miguel Pinho, Instituto Superior Técnico

Abstract—This project report details the design and implementation of a simple RISC processor, both in a single cycle and pipelined versions. In both cases the processor was implemented in VHDL and tested using Xillinx's Vivado simulation software.

Index Terms—Computer Architectures, Pipeline, RISC.

1 Introduction

In this small project it is requested for the students to understand a given computer architecture, and implement some features on it, first in a single cycle processor and then in a five staged pipeline processor.

2 Datapath

2.1 Instruction Decoding outputs

By analyzing both the schematic provided in the assignment and the VHDL code of this same implementation. It was possible to identify what each of the Instruction Decode outputs were used for.

- AA - Selects Register to fetch to and put on signal A;
- MA - Selects Operand A to enter the ALU, will be either signal A or the KNS
- BA - Selects Register to fetch to and put on signal B;
- MB - Selects Operand B to enter the ALU, will be either signal B or KNS
- KNS - It is 32 bit signal that can be selected to enter the ALU or Branch Control Unit;
- FS - Selects what operation is performed by the ALU;
- PL - Defines whether it is a control instruction or not;
- BC - Only meaningful in control instructions. The first bit defines if Link is done or not, the rest of the bits select the condition to be true for the branch/jump to happen;
- MMA - Selects the memory address to be read or written;
- MMB - Select what data should be written to memory;
- MW - Defines whether data is going to be written to memory or not;
- MD - Define what is going to be written into the register file. It will be either the ALU or Memory output;
- DA - selects which register is going to be written to (DA 0 means don't write);

On Section 3.1 it is going to be explained more in detail the state of this outputs for every Operation.

2.2 Arithmetic Logic Unit

One of the most important signals is the FS since it will be used in every cycle since data always flows through the ALU. On table 1 there is a table that shows the operation that is output by the ALU for every FS.

TABLE 1: FS Operations

FS	operation	unit used
0000	A+B	arithmetic
0001	A+B+1	arithmetic
0010	A-B-1	arithmetic
0011	A-B	arithmetic
0100	A and B	logic
0101	A nand B	logic
0110	A or B	logic
0111	A nor B	logic
1000	A xor B	logic
1001	A xnor B	logic
1010	LSL B	shift
1011	LSR B	shift
1100	ASHL B	shift
1101	ASHR B	shift
1110	ROL B	shift
1111	ROR B	shift

Its important to note that this ALU only operates over 32-bit Integer values, so there will be no issues with normalizing and denormalizing values if they were floating point single precision. Also, this ALU only support a handful of instructions, it doesn't have integer division nor multiplication (multiplication could only be implemented by running several ADD intructions for example).

2.3 Structural Hazards

After analyzing the VHDL and the instructions presented on the project statement it can be understood that only the jump and branch instructions are not supported by the original architecture. This is because of two reasons. The first one being that the Instruction Decode Stage is currently not outputting any signal that differentiates between a branch and a jump (if some value is summed to the PC, or if the PC is set to a new value), it only outputs PL which tells if it is a control instruction or not. The other issue is that, there is no way to link. Meaning If there is a jump/branch with link there is no way to save the PC+1 on R15.

To combat the first problem we propose expanding the PL into a 2 bit signal so that we can differentiate between jump and branch:

TABLE 2: PL Operations

PL	Description
0-	No branch/jump
10	Branch Instruction
11	Jump Instruction

In this implementation the MSB of this new PL would have meaning of the old PL and the LSB would be 0 if the instruction was a branch, or 1 if it was a jump (it would be don't care for other instructions). This will be further explained on section 3.1.

To solve the second issue the proposed change is to have a LinkEn signal which would be output by the Branch Control Unit whenever it occurred a jump/branch and it was a linked one. This signal would then be used on the WB stage. If it was 1 the value of PC+1 would be loaded into R15, else the Write Back stage would behave as before. This implementation can be seen on Fig.1.

The Branch Control Unit is explained in detail in 3.2

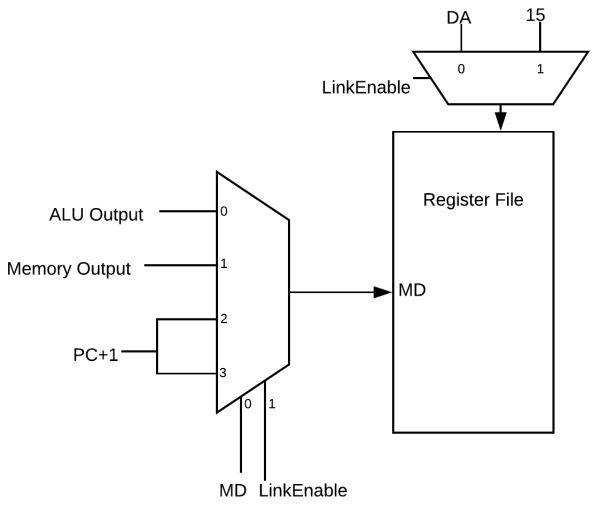

Fig. 1: Computer architecture

3 Architecture Updates

To make this single cycle processor work, was then needed to describe the branch control unit circuit, apply the proposed changes in subsection 2.3 and also program the instruction decode stage.

3.1 Instruction Decode Stage

For the processor to work properly, all the 32 instructions were decoded according to the Table 3. Some of the rows of this table are internal signals that will then be converted into the Instruction Decode Outputs. (Note that although some of the signals on the table here are Don't Care, in the code they are 0s as to avoid some vivado limitations). In this decoding, the PL signal has already been adapted to have 2 bits as described in section 2.3.

First of all its important to note that the value on this table are adapted for both the pipeline and single cycle processors. For instance for shift operations Operand A is actually a don't care on single cycle, but we are still forcing it to be 0 in order to prevent unexpected hazards on the pipeline implementation.

Also, some of the outputs of the instruction decode memory are not the outputs of the ID stage, and others don't even have a relation with this outputs like the BC.

- MASel and MBSel - These signals work in the same way, the LSB corresponds to the MA and MB respectively. The MSB corresponds to whether the register to fetch is the one specified on the instruction or an Harcoded one expressed in dAA or dBA.
- dAA, dBA – Are hardcoded values of the registers to fetch. Apart from one instruction where SB is by default R0 (Jump), these signals were used to set the fetch register to 0 when either RA or RB are not used on the instruction, instead of just using a random register. The reason for this is explained later in this report, for optimizing the pipelined implementation.
- MDSel has a similar meaning to MASel and MBSel, where the LSB corresponds to MD and the MSB defines whether the register we are are going to write back to is the one specified in the instruction or an hardcoded one, dDA. The dDa is used in the Store, Branch and Jump instructions, where the the destination register is set to 0 to assure that no value is written into the register file.
- KNSSel - this is a code that defines what bits correspond to the KNS in the instruction, and if this KNS will be extended into 32-bit using Signed or Unsigned extension. This is important due to different instructions having KNS in different places and with different sign extensions.
- The only signal that is not decoded by this instruction memory is the BC(Branch Control) which is a 4-bit signal where the MSB define if the branch/link is extended or not, and the remaining 3 bits specify the condition to be true in order for the branch/jump to happen. This 4 bits will always correspond the 25 down to 22 bits of the instruction. This can be done because this signal is only meaningful in branch instructions.

3.2 Branch Control Unit

The branch control unit is implemented by the circuit seen in figure 2.

The PCValue that should be loaded onto the PC is either the PC + AD in the case of branches, or just the AD in the case of jumps.

The PCLoadEn signal which defines whether this PCValue should be loaded or not, uses a Mux to select what condition needs to be met and it only outputs HIGH if the Unit is enabled and the PL(1) is active. In the case of single cycle the unit will always be active, and for the pipeline it is only disabled during NOPs).

The LinkEn only outputs HIGH if the PC Loading is enabled and if this instruction is a Linked One.

TABLE 3: Instruction Decoding

Opcode	Mnemoic	PL	dAA	dBA	dDA	FS	KNSSel	MASel	MBSel	MMA	MMB	MW	MDsel
000000	ADD, NOP	0-	-	-	-	0000	-	00	00	-	-	0	00
000001	ADDI	0-	-	0000	-	0000	001	00	11	-	-	0	00
000010	SUB	0-	-	-	-	0011	-	00	00	-	-	0	00
000011	SUBI	0-	-	0000	-	0011	001	00	11	-	-	0	00
000100	AND	0-	-	-	-	0100	-	00	00	-	-	0	00
000101	ANDIL	0-	-	0000	-	0100	101	00	11	-	-	0	00
000110	ANDIH	0-	-	0000	-	0100	111	00	11	-	-	0	00
000111	NAND	0-	-	-	-	0101	-	00	00	-	-	0	00
001000	OR	0-	-	-	-	0110	-	00	00	-	-	0	00
001001	ORIL	0-	-	0000	-	0110	100	00	11	-	-	0	00
001010	ORIH	0-	-	0000	-	0110	110	00	11	-	-	0	00
001011	NOR	0-	-	-	-	0111	-	00	00	-	-	0	00
001100	XOR	0-	-	-	-	1000	-	00	00	-	-	0	00
001101	XNOR	0-	-	-	-	1001	-	00	00	-	-	0	00
001110	SHL	0-	0000	-	-	1010	-	10	00	-	-	0	00
001111	SHR	0-	0000	-	-	1011	-	10	00	-	-	0	00
010000	ROL	0-	0000	-	-	1110	-	10	00	-	-	0	00
010001	ROR	0-	0000	-	-	1111	-	10	00	-	-	0	00
010010	SHLA	0-	0000	-	-	1100	-	10	00	-	-	0	00
010011	SHRA	0-	0000	-	-	1101	-	10	00	-	-	0	00
010100	LD	0-	-	-	-	0000	-	00	00	11	-	0	-1
010101	LDI	0-	-	0000	-	0000	001	00	-1	11	-	0	-1
010110	ST	0-	-	-	0000	0000	010	00	-1	11	10	1	10
010111	B,BL .cond	10	-	-	0000	0011	011	00	00	-	-	0	10
011000	BI,BIL .cond	10	-	-	0000	0011	011	00	-1	-	-	0	10
011001	J, JL .cond	11	-	0000	0000	0011	100	00	10	-	-	0	10
011010	JI, JIL .cond	11	-	-	0000	0011	011	00	-1	-	-	0	10

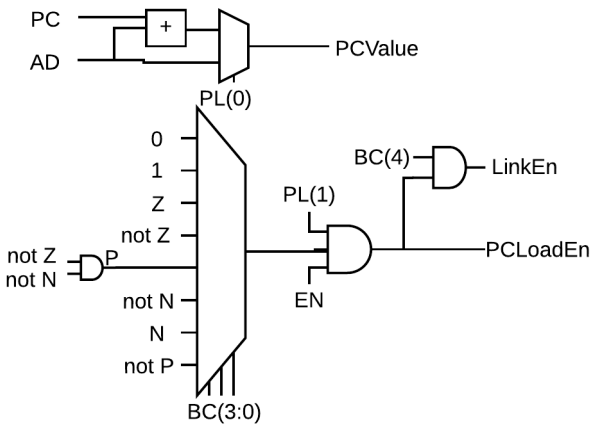

Fig. 2: Branch Control Unit

3.3 Validation

In Order to show that all the instructions are working, it was made a small program, that tests all the instructions in the single cycle version of the processor. Also to assure that the branch control unit was working properly, every condition was tested.

After analysing the simulation it was confirmed that everything was working properly.

4 Pipelined Execution

To turn this single cycle processor into a Five staged pipeline one, state registers were inserted between each of the 5 stages holding the value of the signals that are transmitted between them. It is important to note that due to the alterations made to the processor to support links, some changes must be done to these registers. Since now the value of PC+1 is being propagated from the Instruction fetch to the Write back stage, it should be added another 32bits in these registers, for this signal to be propagated synchronously. The same should me done for the LinkedEn signal which goes from the Execute to the WB stage.

4.1 Hazard Identification

After these change are made it will be possible to analyze existing data and control hazards. Control Hazards will occur every time there is a branch or a jump, Data hazards will occur every time there is a dependency that is not being met.

In the Assembly code below will trigger those two type of hazards.

```
BEGIN:  ADDI R2, R2 , #5
        ADD R4, R2,R2
        ADDI R5, R0, #200
        BR.eq R2 , #1000 , BEGIN
        ADD R2, R0 R0
        NAND R3, R0
```

There is a control hazard on instruction `BR.EQ R2 1000 BEGIN` because this is a Branch instruction. By using the unchanged pipeline code available, the 2 instructions after the branch will always be executed, even in the case where a branch. In this architecture the new address is loaded in the end of the EX Stage, therefore 2 incorrect instructions would have already been fetched.
There also is a Data Hazard because instruction #2 uses R2 which was also used to store the result of instruction #1. Because of this Instruction #2 will fetch the initial value

of R2 and not the one obtained in the end o WB stage of instruction #1. In this architecture there may be a data dependencies with up to the 3 previous instructions.

4.2 Hazard Resolution

In order to resolve these conflicts we implemented the following solutions, that prime for the simplicity and effectiveness, without complicating the design too much. Other solutions which would result in better performances are also suggested, but they would have their own drawbacks, starting with higher design complexity.
Both hazards are detected in the ID stage, after the instruction content is decoded. For the Data Hazard the IF and ID stages simply stall while the data is not ready.For the Control Hazards the pipeline issuing always stalls until it is known whether there is a branch or not, and it stall a second time if there is indeed branch/jump in order to load the new value into the PC.
A Data Hazard has priority over a Control Hazard, as a branch instruction itself needs the most recent value of the registers for the correct branch decision to be made.

4.2.1 Scoreboard

To fix this data hazards the implemented solution was to have a shift register bank that works as a scoreboard and stores in which stages there is or there isn't an operation being done on every register (See Fig.3). So, there will be for the EX, MEM, WB stages one bit for every Register.
The scoreboard uses the id of the register to write to and decodes it generating a one hot signal, pushing a one to one of the bits of the Flag_EX and zero to all the other bits.Also at each clock cycle, the flags propagate from Flags_EX to Flags_MEM and from Flags_MEM to Flags_WB. There are two special cases, the first one is the flags for R0 which are always zero, in fact the R0 flags are not a register, bit just ta wire that is always LOW. The second case are the R15 flags, where the Flag_MEM also takes into account if link is enabled, as to set the R15 to being on stage MEM of the pipeline in the case that the branch control unit detects a branch/jump that happens and that it has link. Although it is not on the schematic, if the ID stage is not enabled (due to some stall or nop), the values pushed onto the flags are 0, since no operation will be done thus no Register will be written to.
Using this scoreboard it is possible to understand if there is a dependency or not with the new instruction decoded in the ID stage just by checking if that instruction uses a register that is at HIGH in one of its 3 corresponding bits on the scoreboard, this detection and resolution is done on the hazard unit where if a conflict is detected, the IF and ID stages will be stalled until the conflict in the later stages is solved, issuing nops to EX as needed.
The fact that reads from register R0 cannot generate data hazards is very important to guarantee that unnecessary stalls are not issued, and that is why in order to prevent such situations, on the ID stage the unused operands are hardcoded to fetch R0.

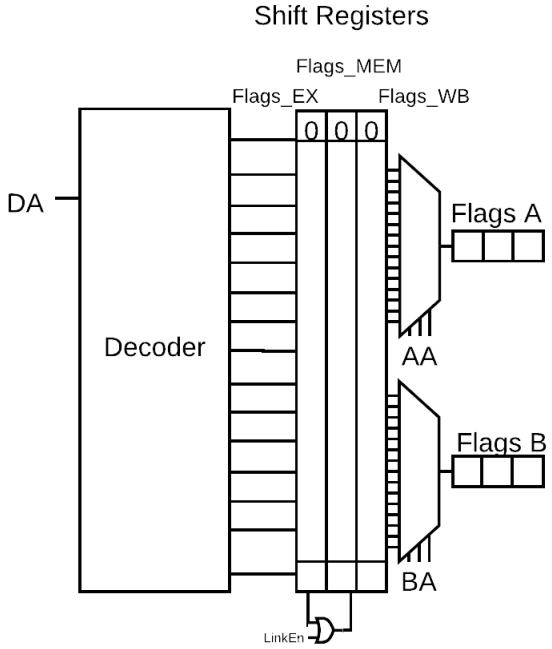

Fig. 3: Scoreboard

4.2.2 Hazard Unit

This unit works is a state machine that manages what stages of the pipeline and on what cycle should they be disabled (See Fig.4). It receives a few inputs:

- Register A usage on the current ID stage;
- Register B usage on the current ID stage;
- PCLoadEnable;
- PL(1);

It starts by setting that there is a Data Hazard if RA or RB are being used in any of the EX, MEM or WB stages and is only active if there is currently not a Nop ocurring on the ID stage so as to prevent the detection of data hazards when the instruction is actually a Nop. The Control Hazard on the ID stage happens when the PL(1) of the ID stage is HIGH meaning the current operation is a branch or jump. Again this only happens when the instruction on ID is not a nop the same reason as before. The Control Hazard on the EX stage is On if the branch/jump condition is met and the EX stage is not performing a nop right now. These 3 signals (DataHazard, ControlHazard_ID and ControlHazard_EX) are the base for the creation of all hazard control signals.

If the DataHazard signal is HIGH the IF and ID stages are stalled (Stall IF and Stall ID are set to HIGH) and a Nop is issued to the next EX stage (Next_Nop_EX to HIGH). While the ID stage is stalled the ScoreBoard flags are being filled with 0, so the DataHazard signal will eventually clear (after 3 cycles at most).
If ControlHazard_ID is HIGH and there is no Data Hazard at the ID Stage a Nop is issued to the next ID Stage (Next_Nop_ID to HIGH, only if DataHazard is LOW) and the IF Stage is stalled (the same thing will happen in the

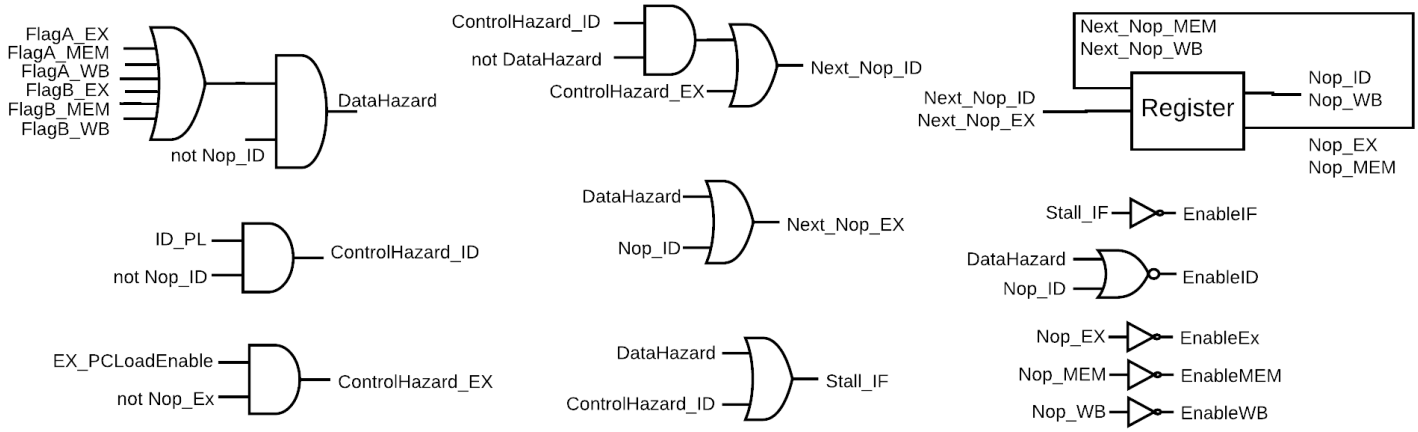

Fig. 4: Hazard Resolve Unit

DataHazard, so no need to make sure it is LOW). The branch instruction is allowed to progress in the pipeline, but the next instruction is not fetched, instead a nop takes its place. If DataHazard_EX is HIGH, another nop is issued to the ID stage, has the new PC address has to be loaded before a new address can be fetched. The IF stage is allowed to continue, has the new pc must be loaded.

The outputs of the Hazard Unit are the enables for the five stages. These are generated based on the stalls and nop signals. A stage is only enabled if there is not a nop nor a stall in it. The nop signals are propagated internally synchronously along the sequence of the pipeline ($ID \rightarrow EX \rightarrow MEM \rightarrow WB$), using a shift register.

An example for each hazard is present in Fig. 5 and Fig. 6.

If a data hazard was detected, the current IF and ID stages will be stalled, as not to advance the PC to the next cycle and to keep the current ID instruction from executing before its operands are ready. This will keep on happening has long as the hazard is occurring. Also on the next cycle, there will be a nop on the EX that will be propagated to MEM and an then to the WB stages as the cycles pass. With this implementation, if a hazard occurs due to an instruction needing a value from the instruction that happened immediately before, there will be a data hazard for 3 cycles and thus the process described above will happen 3 times.

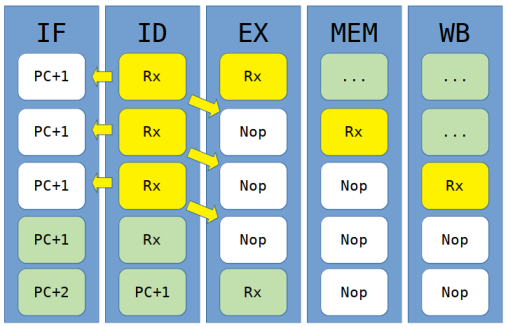

Fig. 5: Data Hazard example

For control hazards as soon as the instruction is decoded, it will be set to be a nop on the next cycle in the ID stage so that the branch is not decoded again. Also the IF stage is stalled and the PC is kept unchanged. On the EX stage, after evaluating if the branch/jump actually happens the PCLoadEnable is generated, and in case the control condition was met, then another nop will occur on the ID stage.

So, if the branch occurs, 2 nops will happen, if it does not only 1 nop is needed to solve the hazard.

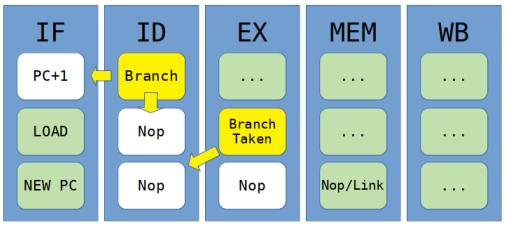

Fig. 6: Control Hazard example

4.2.3 *Validation*

To better explain how it works some instructions were put on the pipeline to show what happens at each cycle and at each stage (See Fig.).

In the figure the first group of 3 stalls is due to a dependency of the Store operation on the ADDIs before it. On the Load instruction the same happens, it needs to stall 3 cycles until the Load for the same reason. On cycle 13 because there is a branch instruction, there will be a stall on the next cycle. Since the branch doesn't occur, no more stalls are issued. Finally when the Instruction JIL is decoded an stall is issued, and in the next cycle when it is verified that the branch happens, another stall e issued, so that the pc has time to be loaded.

4.3 Performance Impacts

Performance wise its first important to see the speed up relative to single cycle. This program take 399 cycles to run the teacher's testing program (before going onto the infinite

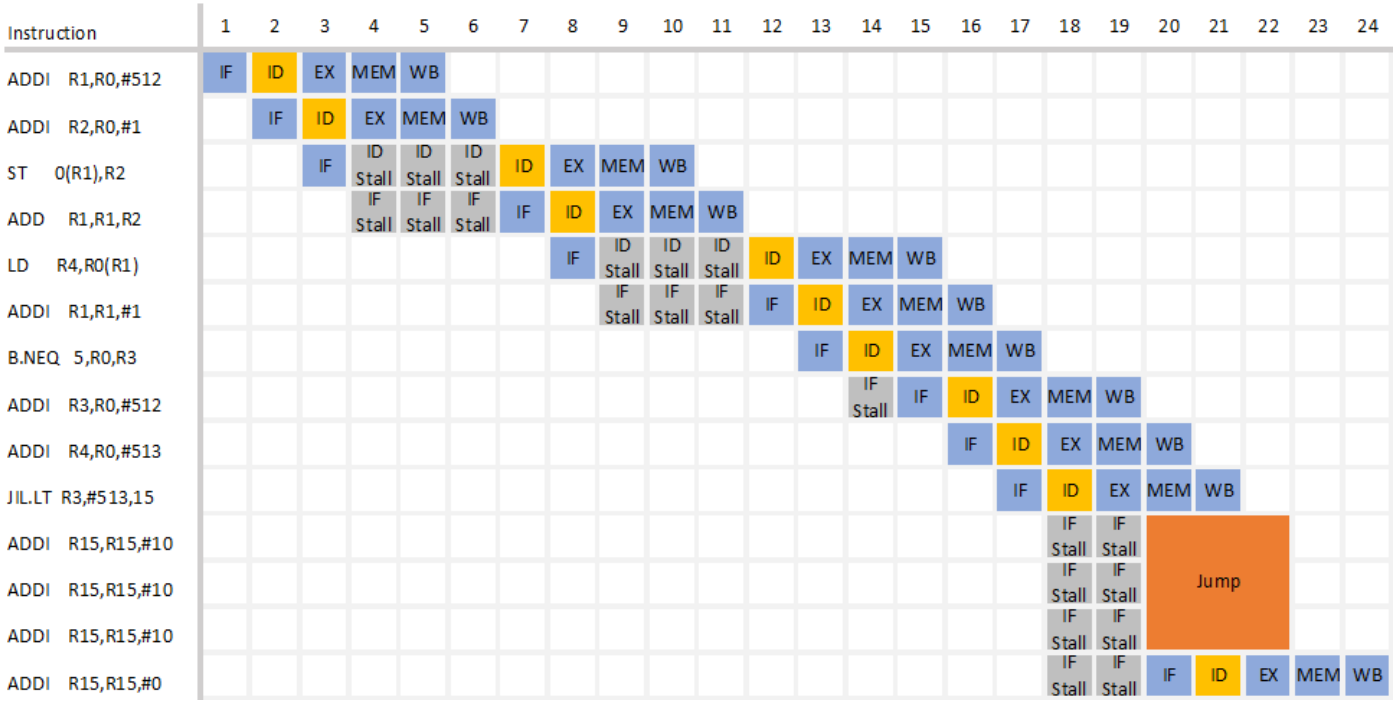

Fig. 7: Cycles over time

loop). The single cycle processor takes 162 cycles. Using the speed up formula

$$\frac{\#Cycles_Single}{\#Cycles_Pipelined} \cdot \frac{CLKfreq_Pipeline}{CLKfreq_Single}$$

.

Admitting the the pipeline stages are not equally divided, lets stay the pipeline frequency is only 4 times higher than in single cycle instead of the ideal 5:

$$Speedup = \frac{162}{399} \cdot 4 = 1.6$$

.

This speedup is not very impressive since with quite a bit more area used the result is only 60% better, where in theory the maximum increase (without delays) is 300% (taking into account a 4 times higher clock cycle).

In practice, the number of nops issued is not the only factor in a performance decrease, in practice. The extra logic needed to address the hazards will increase the power spent and heat to dissipate and increase the critical paths, and thus reduce the frequency. In our solution, the worst increase in critical path seems to be that that the signal to decide if there is a control hazard in the EX has to wait for the generation of the flags and propagate in the branch control unit, and the enable signals still have to be propagate before the end of that cycle. The ScoreBoard also implicates quite some extra logic, including flip-flops. However, the advantage of this simple solution is that this overhead is not that significant.

4.4 Other solutions for the hazards

The performance could be increased if more elaborate solutions for the the hazards were implemented. The delays in the Data Hazard could be completely removed if forwarding paths were to be implemented and selected according to where the needed data is located (which is already encoded in the ScoreBoard flags). If the most recent data was in EX stage or MEM stage, the forward paths could come from the state registers after these respective stages to a multiplexer at the beginning of the EX stage. If it was instead int the WB stage it could be forwarded right to that same ID stage from the WB register state output. Even more, if the WB stage was changed to write to the register file on the falling edge (if half the clock period was enough for the write back critical path), this last forward path would not even be needed.

For the control hazards, a solution with less delays would be to have branch predict not taken, where when the branch condition is false there is no nop (in our solution 1 nop is always issued). Instead of stalling the IF and issuing a nop to the ID stage when the control hazard is detected, the next instruction would always be allowed to continue. In case the branch turned out to happen, that instruction would just have to be marked as a nop.

However, more complex solutions implicate higher penalties in the increase of critical path and circuit size. With the forwarding paths, for example, the extra multiplexer in the EX stage (or the increase of the size of the already existing ones) would increase the critical path in the stage which probably is already the biggest one. Therefore each of this design choices has to take into consideration these pros and cons, for each particular architecture.

5 Conclusion

In conclusion we found the resulting performance of our pipeline implementation not very satisfactory, but still it was very interesting to deal with all the nuances that must be resolved on the pipeline processor for it to work correctly, and that are not as simple as they first appear.