Lab #3: Pipelined Processor

CEG 3156: Computer Systems Design

Winter 2021

School of Electrical Engineering and Computer Science

University of Ottawa

Professors name:

Dr. Rami Abielmona

Student Name and Student #: Ritvik Johar, 300074686
Student Name and Student #: Gianluca Coletti, 300065278

Table of Contents

List of Figures

List of Tables

1.0 Introduction

1.1 Purpose

The purpose of this lab is to design and implement a pipelined RISC processor. Through this lab we are able to understand pipelined RISC processors and containing hazards.

All circuits are programmed in very high speed hardware description language (VHDL) at the structural level.

1.2 Problem

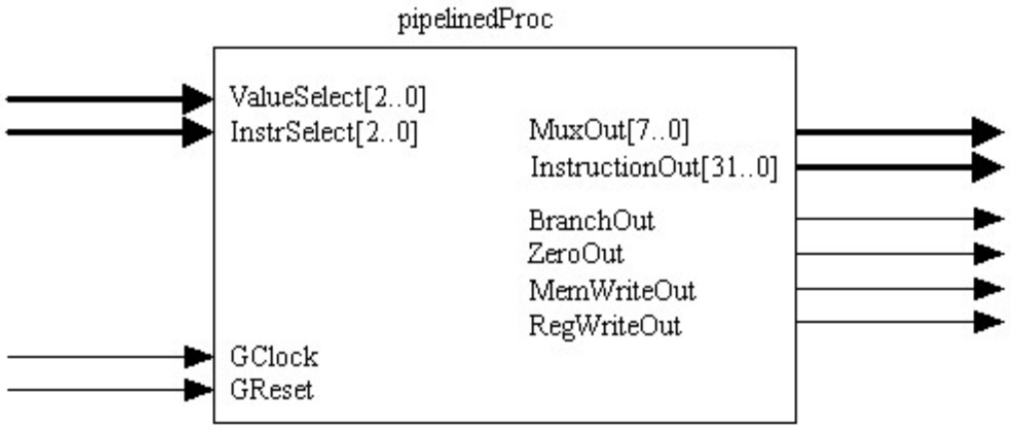

Figure 1: Pipelined processor system entity
Accessed from Laboratory #3: Pipelined processor, R. Abielmona

2.0 Concepts of pipelining

2.1 Introduction

The pipeline is to break down a repetitive process into several sub-processes, each of which runs in parallel with other sub-processes, this working method is very similar to the production line in the factory. Essentially, pipeline technology is a time-parallel technology. In the pipeline processor, continuous tasks are one of the necessary conditions to fully utilize the efficiency of the pipeline. Four characteristics of the instruction set are beneficial to pipeline execution:

- Keep the length as consistent as possible , which is helpful to simply fetch and decode the instruction.
  - 32-bit MIPS instruction, convenient for address calculation: PC+4.

- The format is few, and the source register location is the same , which is helpful to fetch the operand when the instruction is unknown.

  - The Rs and Rt positions of the MIPS instruction are fixed, and the values of Rs and Rt can be read when the instruction is decoded.

- Only the load and store instruction can access the memory , which is beneficial to reduce the number of operation steps.

- The execution steps of lw/sw address calculation and operation instruction are organized in the same cycle.

- Aligned storage in memory , which helps reduce the number of fetches.

2.2 Five stages of pipeline

Instruction Type	Pipeline Stage				
Load	IF	ID	EX	MEM	WB
Store	IF	ID	EX	MEM	
R-Type	IF	ID	EX		WB

- IF (Instruction Fetch) : According to the memory address stored in the PC, find the instruction at that address in the memory, and then the instruction is stored in a register. At the same time, the PC points to the next instruction (PC+4).

- ID (Instruction Decode/Register Fetch Cycle) : Operate the instruction fetched from the IF stage. Decode the instruction, and finally find the data stored in the register required by the instruction. If the instruction has only one jump instruction, then at this stage, the acquired values need to be compared according to the meaning of the jump instruction. If the comparison result is true, a jump is performed. If the comparison result is false, no jump is performed, and execute the next instruction; if the instruction needs to fill some bits in the instruction, it is also completed in the ID stage, such as filling the upper four bits to meet the instruction result to 32 bits and calculate the address of the instruction that may jump.

- EX (Execution/Effective Address Cycle) : ALU calculates the result of the ID stage. In the ID stage, the values of the registers required for instruction calculation have been fetched. Then, in the EX stage, the values of these registers need to be calculated

according to the meaning of the instructions. The calculation varies according to the instruction. There are mainly three types of ALU calculations:
- ALU calculates the effective address unit based on the address supplemented in the ID, and finally fetches the required memory address;
- According to the meaning of the instruction, the value fetched from the register, perform operations, such as adding the values of two registers;
- Calculate the immediate result based on the value of the register and the supplementary value.

- MEM (Memory Access) : If the current instruction is a LOAD instruction, then the corresponding value is fetched from the memory according to the memory address calculated by EX; if the current instruction is a STORE, then the register value is stored according to the memory address and register value calculated by EX into the memory address. Other instructions are generally not designed for memory access.

- WB (Write Back) : Write the calculated final register value to the register file. This operation includes the value fetched from memory and the result obtained by arithmetic operation.

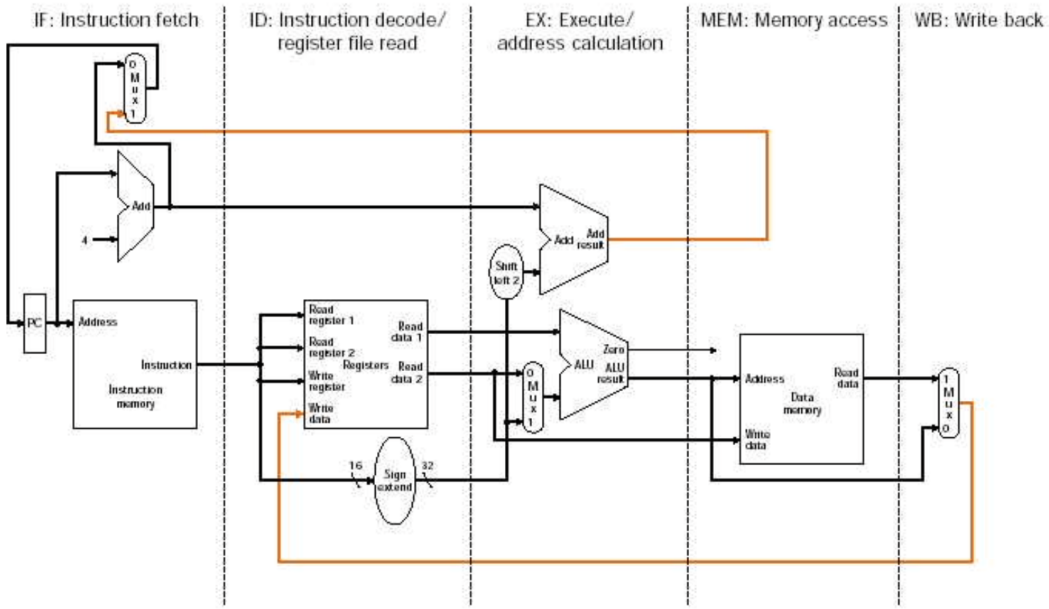

Figure 2: Single-cycle processor divided into 5 stages
Accessed from Laboratory #3:Pipelined processor, R. Abielmona

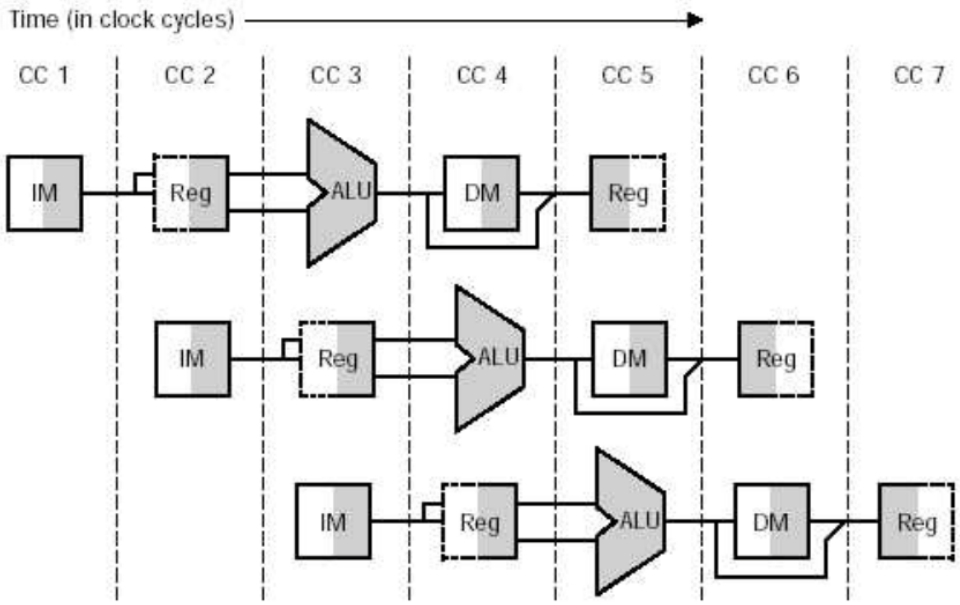

Figure 3: Multiple instructions running on same datapath
Accessed from Laboratory #3:Pipelined processor, R. Abielmona

2.3 Pipeline Hazards

Pipeline hazards refers to that for a specific pipeline, the next instruction in the instruction stream cannot be executed at a specified clock cycle due to the existence of the correlation. There are three types of pipeline hazards: structural hazards , control hazards and data hazards .

- Structural Hazards : Use a function part for multiple instructions at the same time. The following figure X has this hazard, the two instructions try to write back to the register at the same time.

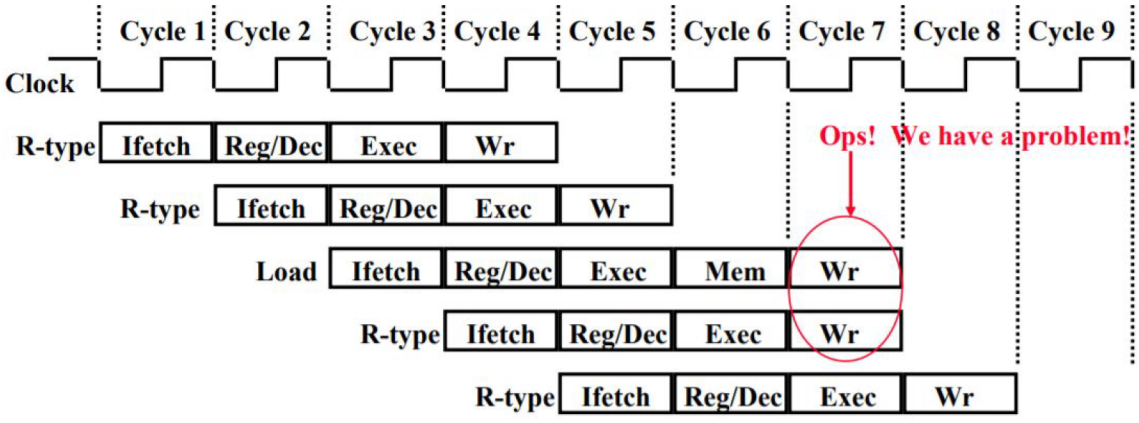

Figure 4: Example of structural hazard
Retrieved from google

- Control Hazards : The pipeline encounters hazards caused by branch instructions and other instructions that change the value of the PC. For branch instructions, there are two consequences. One is that the branch is "successful" and the PC changes to the branch target address. The other is "failure", which means that the PC value keeps increasing normally and the next instruction is executed in the original order. If the branch is successful, the PC value is changed only after both the condition determination and the branch address calculation are completed.

- Data Hazards : Hazards that occur when instructions are executed in the pipeline overlapping the execution results of previous instructions. The following figure has this hazard: The correct data of register R1 is generated only after the first instruction is executed to the fifth cycle, and the following four instructions use the R1 data that is generated until the fifth cycle, but some instructions (like 2nd, 3rd, 4th instruction) require the use of R1 value before the fifth cycle, for example, the subtraction instruction of the second instruction needs to use the data of R1 in the second cycle), which is obviously unreasonable and will cause data conflicts.

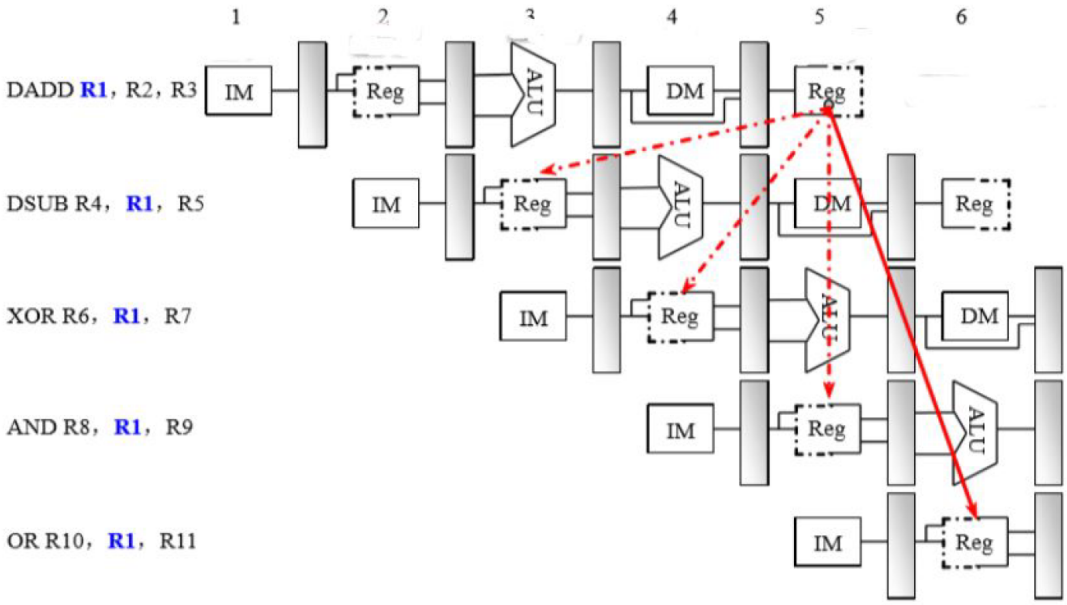

Figure 5: Example of data hazard
Retrieved from google

To solve this situation, some methods can be used:

- The design can be specified: Write data on rising edge and read (fetch) data on falling edge.
- Solve with data-oriented path technology.
- Add pipeline interlocking hardware and insert "pause" (bubble).

2.4 Control signals in Pipeline

Generate all control signals for each stage of this instruction in the fetch / decode (Reg /Dec) stage. When the next clock arrives, pass the execution results and all the data to be used in the later stages (such as: instruction, immediate value, destination register, etc.) and control signals are saved to the pipeline register.

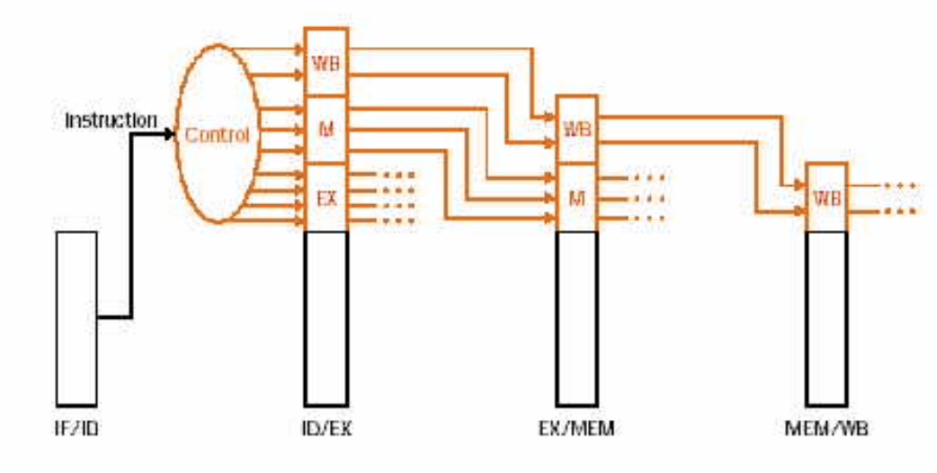

Figure 6: Control signal additions to pipeline signals
Accessed from Laboratory #3:Pipelined processor, R. Abielmona

- IF and ID (Reg/Dec in Figure 2.4.1) stage does not have control signals.
- EXE stage has four control signals.
  - ExtOp (expander operation): 1 = Sign extension, 0 = Zero extension.
  - ALUSrc: 1 = Derived from expander, 0 = Derived from BusB.
  - ALUOp: Main controller output, used to assist local ALU control logic to determine ALUCtrl.
  - RegDst (destination register): 1 = Rd, 0 = Rt.
- MEM stage has two control signals.
  - MemWr: 1 = STORE instruction, 0 = other instruction.
  - Branch: 1 = branch instruction, 0 = other instruction.
- WB (Wr in Figure 2.4.1) stage has two control signals.
  - MemtoReg: 1 = DM output, 0 = ALU output
  - RegWr: 1 = result is register instruction, 0 = result is other instruction

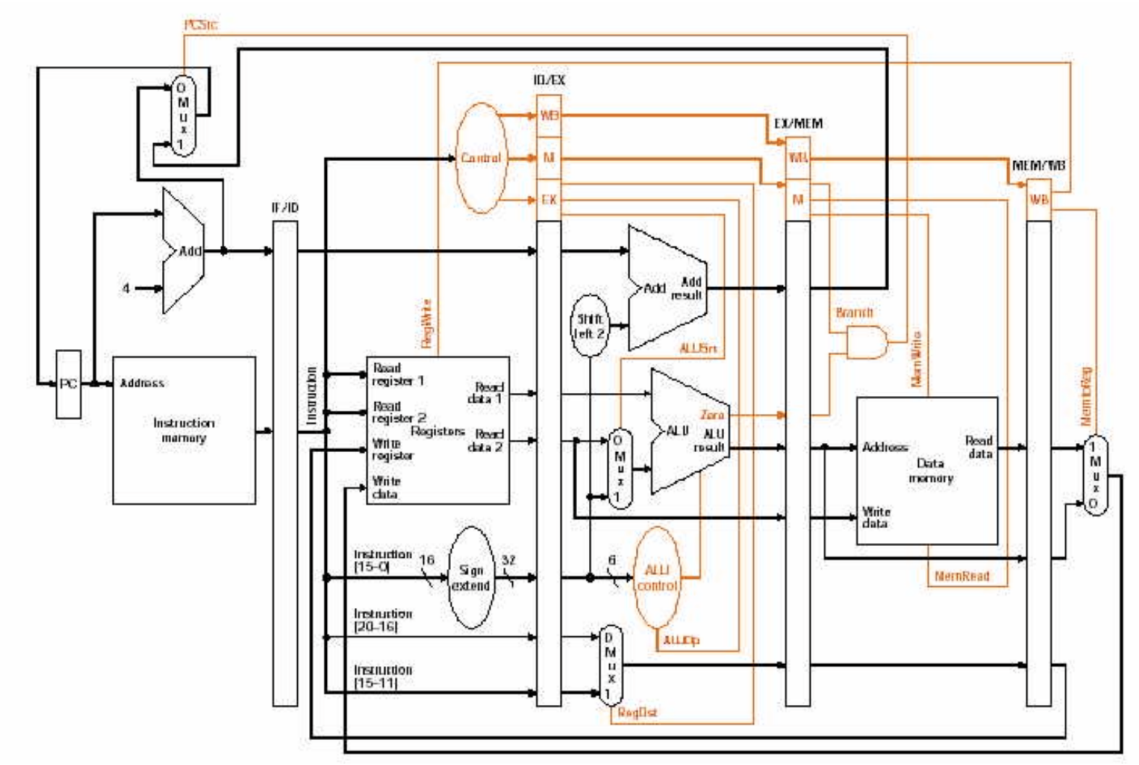

Figure 7: Pipelined MIPS processor with control path
Accessed from Laboratory #3:Pipelined processor, R. Abielmona

3.0 VHDL Solution

The above described data paths were realized in the form of components. Only structural VHDL and RTL logic was used (with some exceptional Behavioural in memory units). To ensure the correct results, each component was simulated and verified. The components created are as follows:

- ALU
- ALU Decoder
- Control Unit
- Instruction Decoder
- Instruction Memory
- 2-to-1 Multiplexer (5 bit)
- 2-to-1 Multiplexer (32 bit)
- 4-to-1 Multiplexer (32 bit)
- PC
- Register File
- Shift Left by 2 bits (SL2)
- Select MUX
- Sign Extend
- Data Memory

- Full Adder (1 bit)
- 4 to 1 MUX (32 bit)
- Ripple Adder (8 bit)
- Ripple Adder (32 bit)
- pipelinedProc (*Top level entity*)
- Zero Extend
- IF_ID pipeline register
- ID_EX pipeline register
- EX_MEM pipeline register
- MEM_WB pipeline register

3.1 Calculation of Max Clock Frequency

The processor has longest delay:

$= lw\,delay = PC + instruction\,mem + register\,file + mux + ALU + data\,memory + mux + regwrite$

$= 1 + 10 + 10 + 2 + 15 + 10 + 2 + 10 = 60\,ns$

Now have additional 2 mux + 1 ALU+ 1 mux = 51+ 4 + 15 + 2 = 81 ns

The fastest CLK Rate = 1/81ns = 12.35 MHz

3.2 Calculation of CPU Execution Time

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

4 load (x5 cycles)
1 sub (x4)
1 or (x4)
3 branch (x3)
2 store (x4)
1 add (x4)
1 jump (x3)
TOTAL = 13

$$\text{CPI} = [(4x5) + 4 + 4 + (3x3) + (2x4) + 4 + 3]/13 \cong 4$$

Thus,

$$\text{CPU time} = (13 \times 4)/ 12.35\times10^{6}$$

$$\text{CPU time} = 4.21\times10^{-6} \text{ seconds}$$

Comparing from single-cycle processor:

$$\text{Single-cycle proc CPU time} = 9.72 \times 10^{-4}$$

Therefore, we have about a **198%** increase in CPU speed compared to the single-cycle processor. This increase in CPU strictly due to the increased CPI as a result of the 5-stage pipeline implementation.

3.3 Calculation of ALU Worst Path Delay

Again, in a 32-bit [ripple carry] adder, there are 32 full adders, so the critical path (worst case) delay is:
$31 \times 2(for\ carry\ propagation) + 3(for\ sum) = 65$ gate delays

Therefore,

$$\text{Worst case delay} = 65 \times 0.01\ ns = \underline{0.65\ ns}$$

3.4 Discussion of Tool

The tools that were used for this lab were ModelSim and Quartus II; which were used in combination with the Altera Cyclone DE2 board. ModelSim is a multi-language HDL simulation environment which allows us to design and simulate the VHDL files. These VHDL files were then compiled in ModelSim and used to help build our VHDL components and create our top level entity for the design. However, this lab did not require a demonstration of our solution with the Altera Cyclone DE2 board but instead verified our results using ModelSim and timing diagrams.

4.0 Verification

4.1 Simulation Results

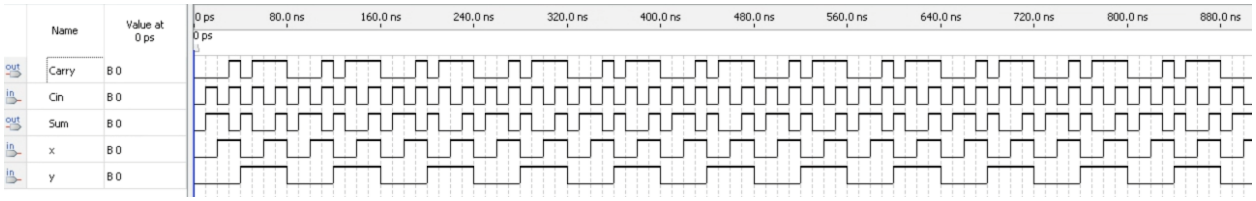

Figure 8: Simulation of full adder (1 bit).

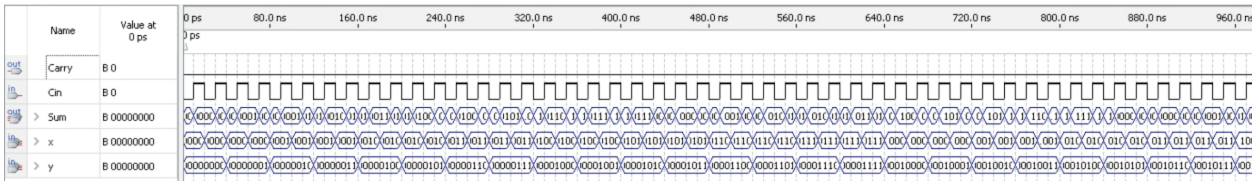

Figure 9: Simulation of ripple adder (8 bit).

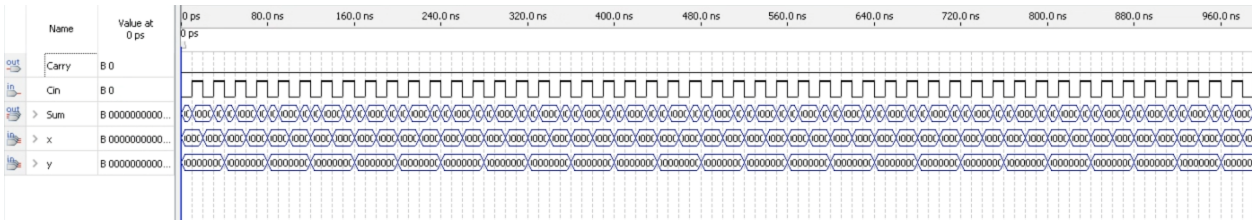

Figure 10: Simulation of ripple adder (32 bit).

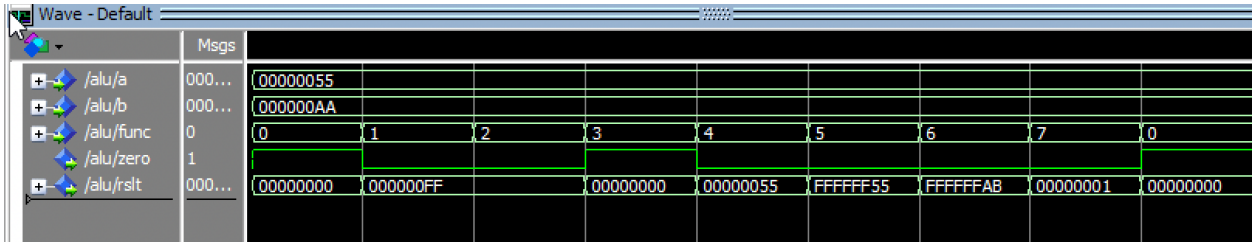

Figure 11: Simulation of ALU unit.

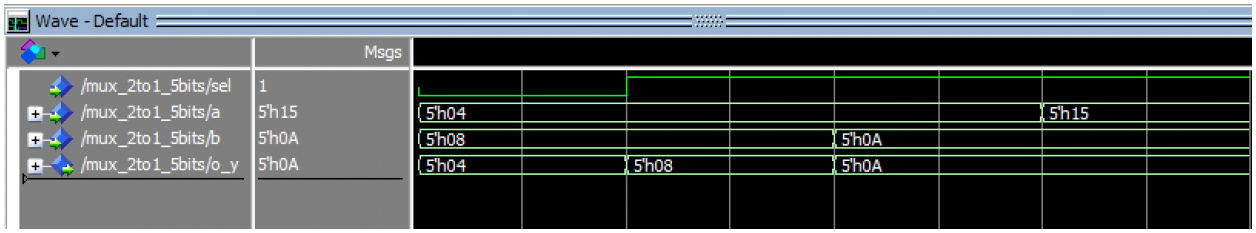

Figure 12: Simulation of MUX 2-to-1 (5 bit).

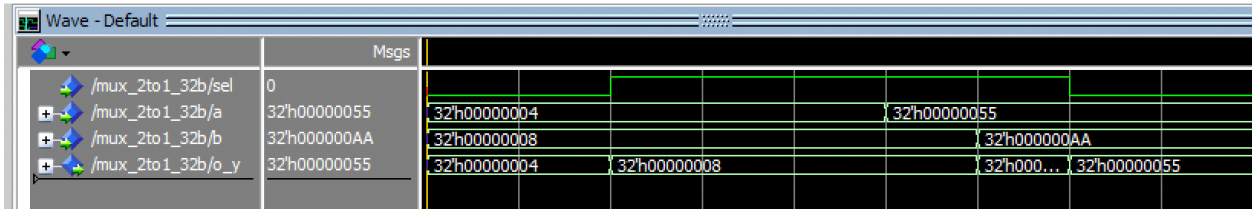

Figure 13: Simulation of MUX 2-to-1 (32 bit).

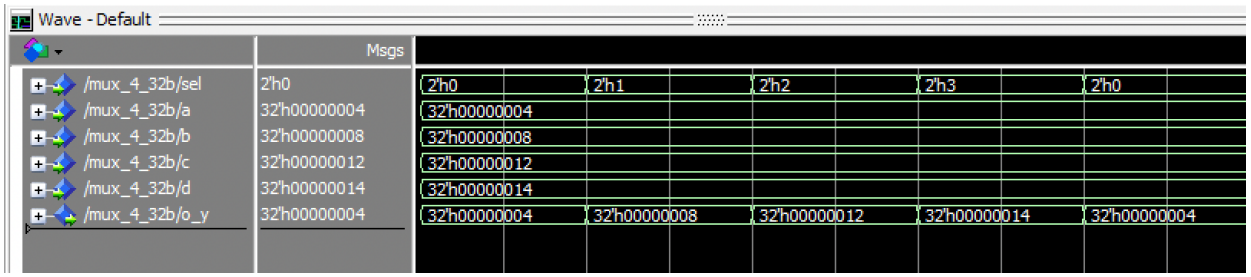

Figure 14: Simulation of MUX 4-to-1 (32 bit).

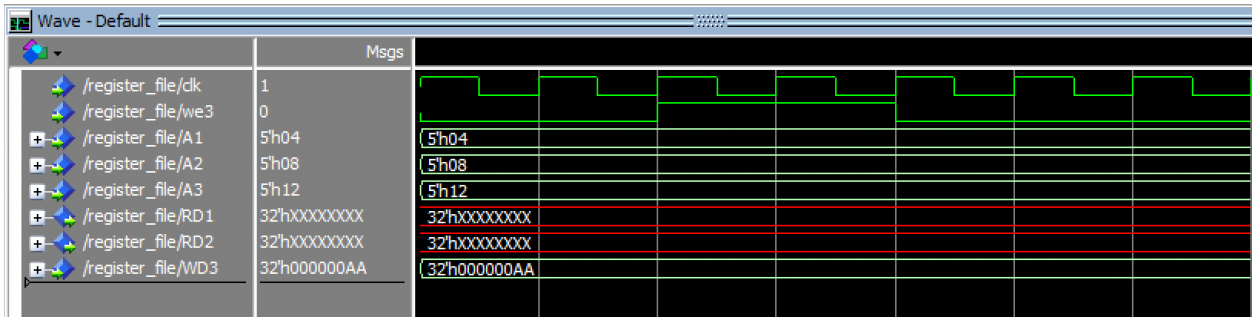

Figure 15: Simulation of register file unit.

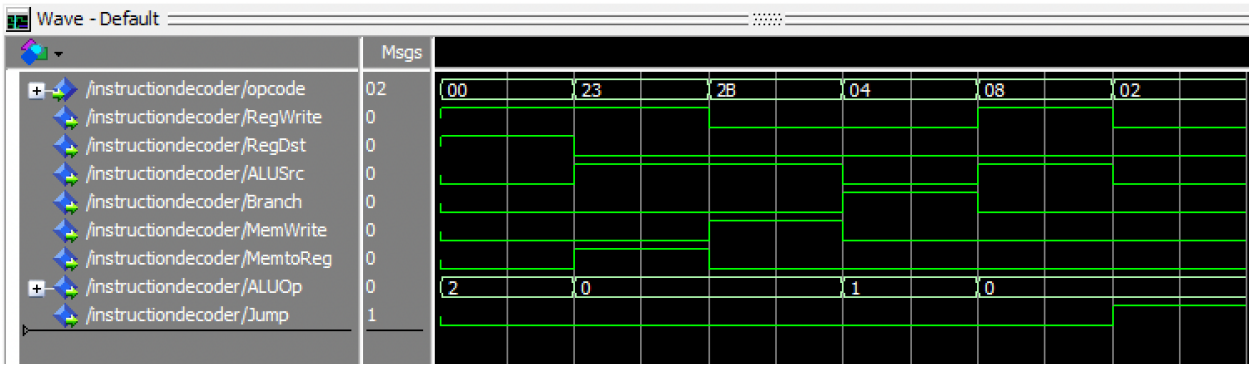

Figure 16: Simulation of instruction decoder.

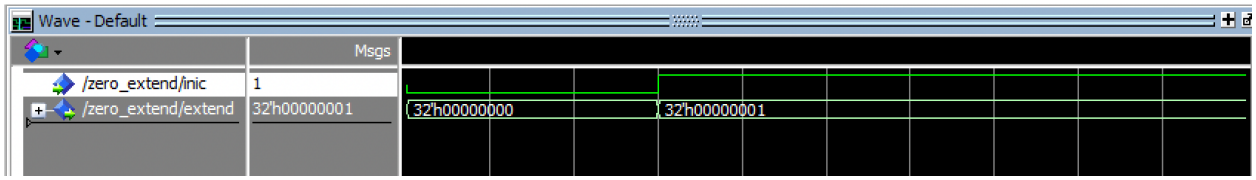

Figure 17: Simulation of zero extender unit.

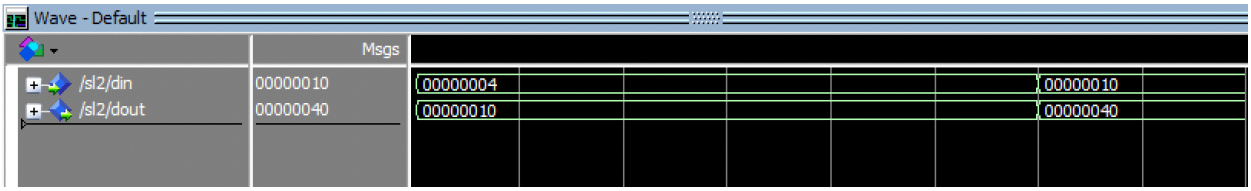

Figure 18: Simulation of arithmetic shift left by 2 bits unit.

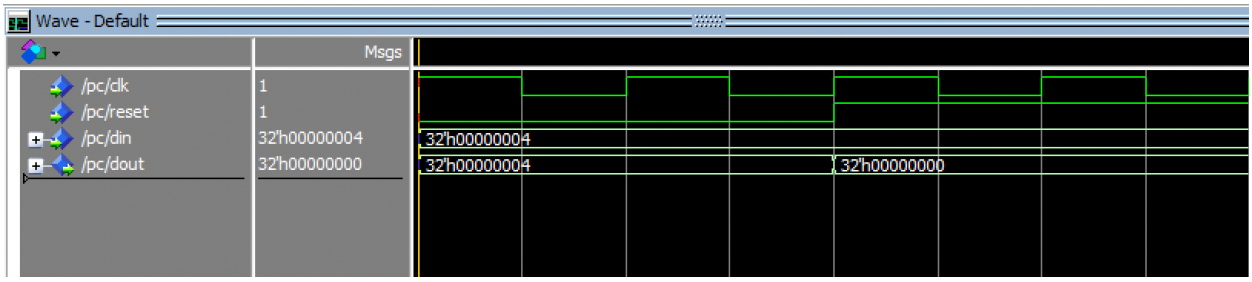

Figure 19: Simulation of PC increment component.

Figure 20: Simulation of the MIPS control unit.

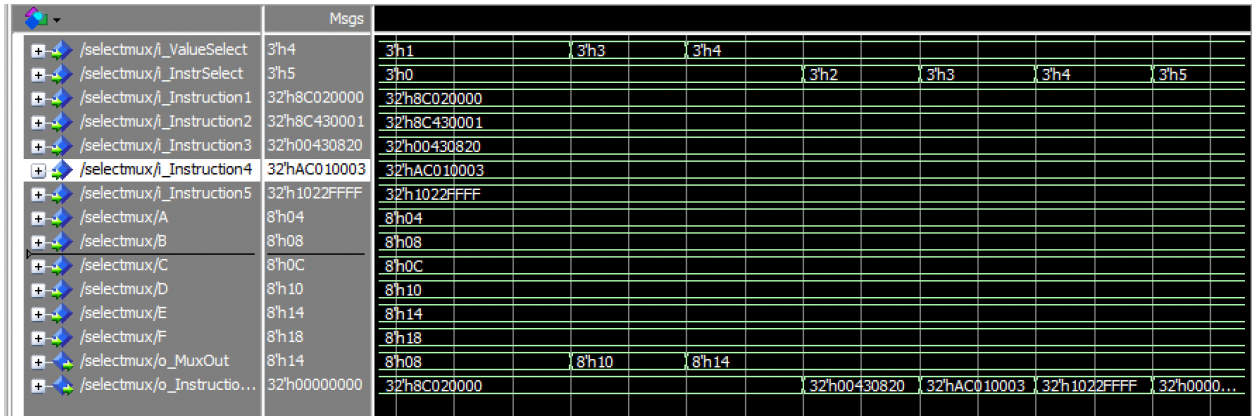

Figure 21: Simulation of new SelectMux.

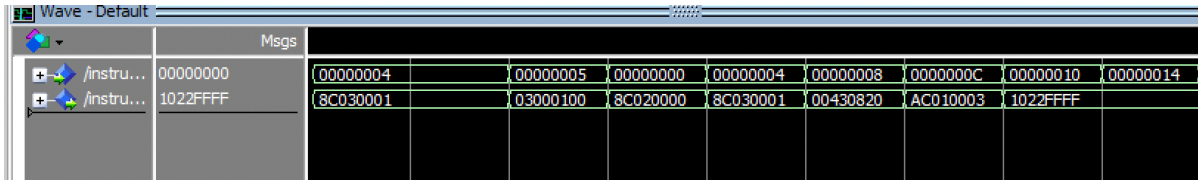

Figure 22: Simulation of instruction memory unit.

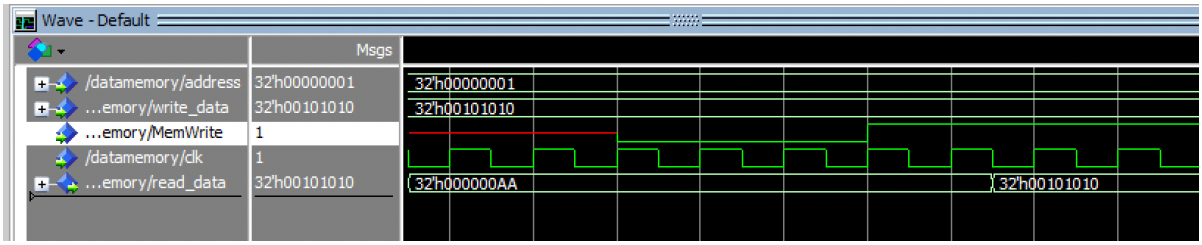

Figure 23: Simulation of data memory file.

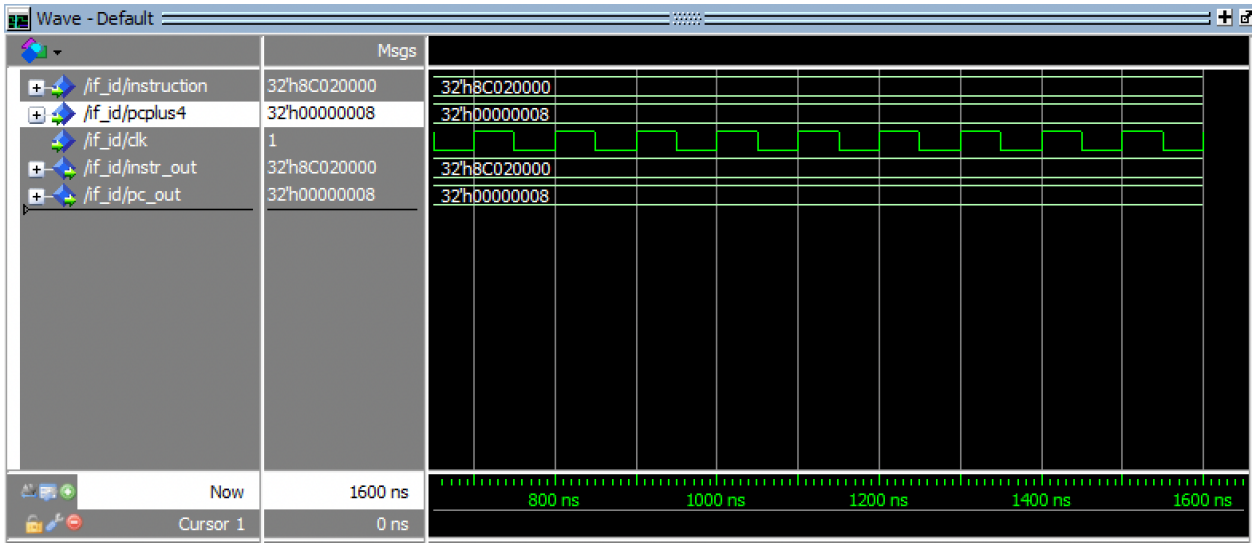

Figure 24: Simulation of IF/ID pipeline register.

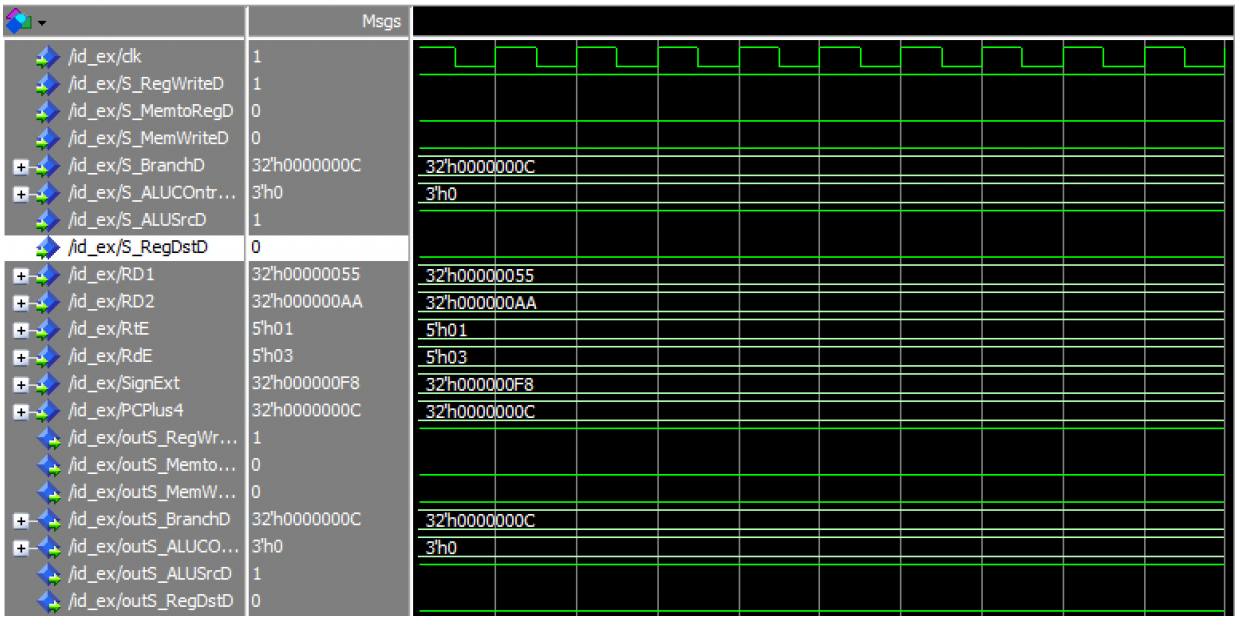

Figure 25: Simulation of ID/EX pipeline register.

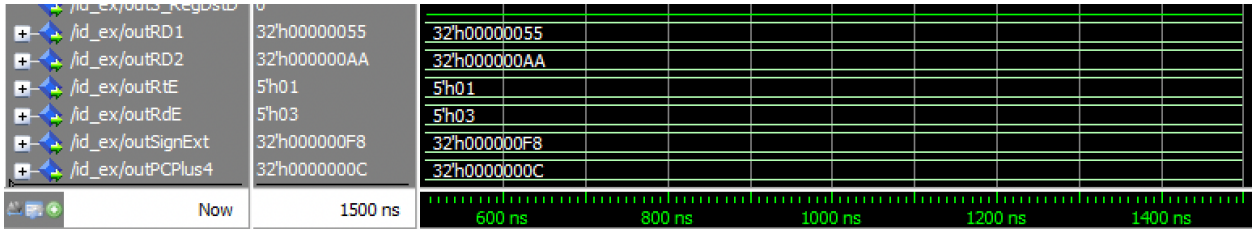

Figure 26: Simulation of ID/EX pipeline register (Cont.).

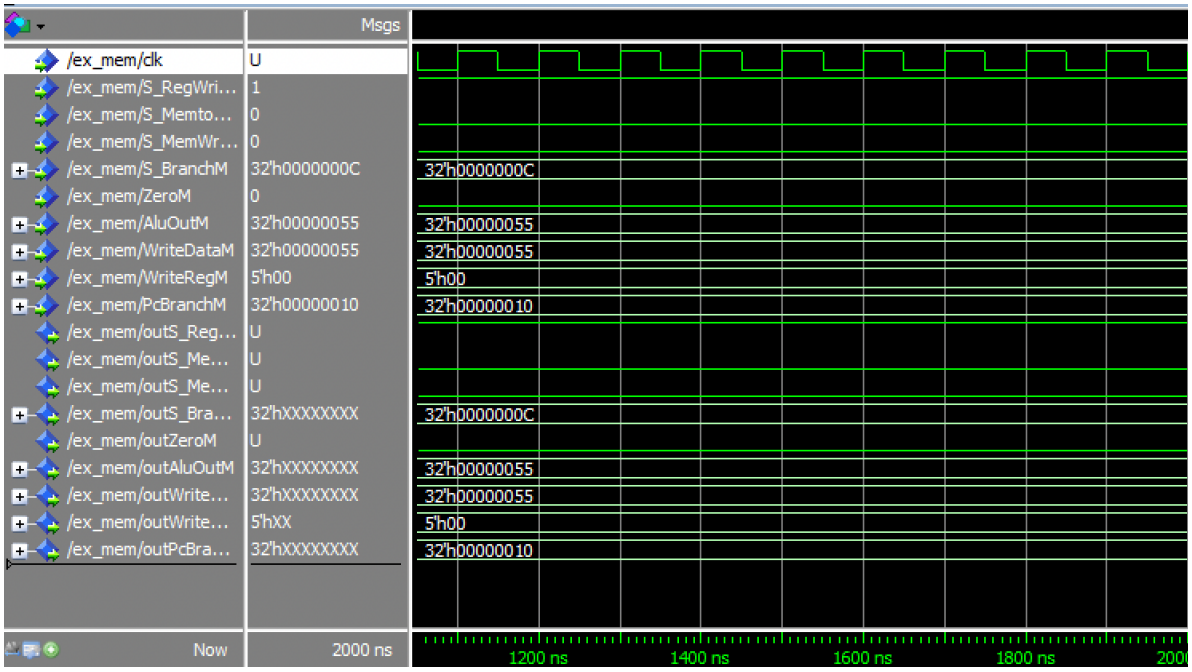

Figure 27: Simulation of EX/MEM pipeline register.

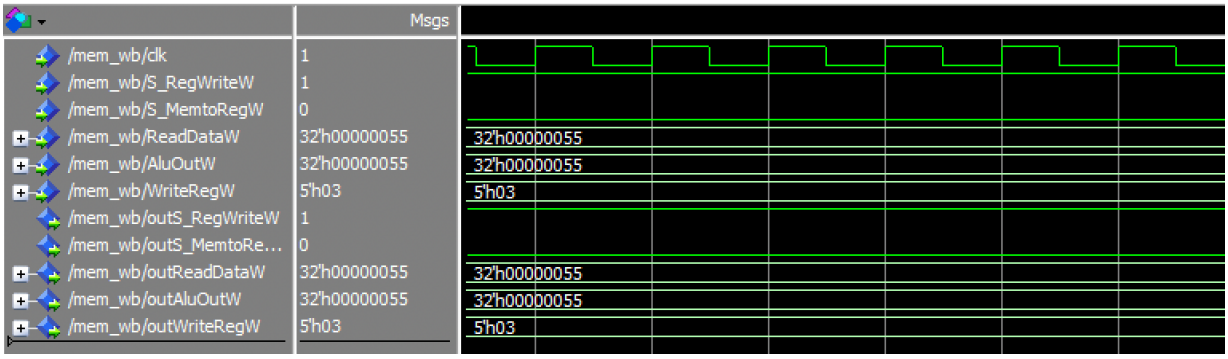

Figure 28: Simulation of MEM/WB pipeline register.

4.2 Discussion

Our main errors pertaining to our design for this lab consisted of the ALU adding the wrong two values denoted by *srca* and *srcb* for the ALU Mux (which was a known flaw in our Single Cycle processor). As well, another issue arised with the simulation of our pipelined registers not correctly sending their control signals to other parts of the MIPS processor. We also did not implement the Hazard and Forwarding unit in VHDL due to the pipeline errors.

5.0 Conclusion

5.1 Summary and conclusion

In this experiment, although we did not solve all the problems we encountered in this lab, we have a complete understanding of pipelined RISC processors. We know the five stages of pipeline, pipeline hazards and control signals in pipeline. Moreover, the datapath diagrams help us have an idea to separate the whole processor into small pieces.

6.0 Prelab

6.1 Hazard unit control logic

The control signal for hazards is:

x = (EX.MemRead == 1) and (EX.Rt == ID.Rs) or ((EX.Rt == ID.Rt) and (Opcode != 001110) and (Opcode != 100011)

Table 1: Truth table for hazard control logic

x	PC.WriteReg	IF/ID.WriteReg	*Stall*
1	0	0	1
0	1	1	0

Thus,

$$\text{PC.WriteReg} = \text{not}(x)$$
$$\text{IF/ID.WriteReg} = \text{not}(x)$$
$$\textit{Stall} = x$$

6.2 Forwarding unit control logic

EX:

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

MEM:

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01