ECE 485 Project #2

Jay Mundrawala

November 22, 2009

Abstract

This project describes the design and implementation of a multicycle MIPS datapath. Along with it are testbenches that issue the appropriate signals for a few signals.

1 Introduction

The goal of this project was to implement a MIPS datapath capable of executing a small set of instructions. For this project, we were to implement the following instructions:

- LW
- SW
- ADD
- SUB
- AND
- OR
- SLT
- BEQ
- J
- BNE
- ADDI
- SLL
- LUI
- JAL

The source code for this datapath is available in the appendix and in a git repository on github.

2 Design

The design for a multicycle MIPS datapath is quite simple. It is shown in Figure 1. The data path has the following signals coming from the control unit, which describe how the datapath should operate: PCWrite, IorD, MemRead, MemWrite, IRWrite, RegDst, MemToReg, RegWrite, ALUSrcA, ALUSrcB, ALUOp, and PCSource. Tables 1 through 6 enumerate the possible values for these signals and Listings **??** contains their definitions.

Signal name	Effect when deasserted	Effect when asserted
PCWrite	None	PC is written
PCWriteCondEq	None	PC is written of zero is asserted
PCWriteCondNEq	None	PC is written of zero is deasserted
IorD	Instruction is Fetched	Data is fetched
MemRead	None	Memory at given address is read
MemWrite	None	Data is written to given address
IRWrite	None	Instruction register is written
RegWrite	None	Data is written to register
ALUSrcA	Port A of ALU gets PC	Port A of ALU gets register A's output

Table 1: Actions of 1-bit control signals

ALUSrcB	
Signal value	Effect
ASB_REGB	Port B of ALU gets register B's output
ASB_FOUR	Port B of ALU gets 4
ASB_SEXT	Port B of ALU gets instruction[15-0] sign extended to 32 bits
ASB_SEXTS	Same as previous except value is shifted left by 2

Table 2: Actions of ALUSrcB signal

ALUOp	
Signal value	Effect
AOP_AND	ALU will 'and' the two operands
AOP_OR	ALU will 'or' the two operands
AOP_ADD	ALU will add the two operands
AOP_SUB	ALU will subtract A - B
AOP_SLT	ALU will output '1' if $A < B$, '0' otherwise
AOP_NOR	ALU will NOR the two operands
AOP_SLL	ALU will shift B left by instruction[10-6]

Table 3: Actions of ALUOp signal

With these signals defined, we can begin by defining each functional block in Figure 1. First, we have the PC. This is the program counter and stores the address of the next instruction to be executed. The memory block is our RAM. For this project, the test benches only output one value when the memory is read, and that is the instruction we are testing. This allows us to test the datapath without creating a memory module. Nothing is ever written to the memory; when testing LW we analyze the value on the

PCSource	
Signal value	Effect
PS_PCINC	Value to PC is $PC + 4$
PS_ALUOUT	Value to PC is that of the register ALU OUT
PS_JMP	Value to PC is $PC[31 - 28] + Instr[25 - 0] << 2$

Table 4: Actions of PCSource signal

RegDst	
Signal value	Effect
RD_RT	Write register is set to instr[25-21]
RD_RD	Write register is set to instr[20-16]
RD_RA	Write register is set to $R31$

Table 5: Actions of RegDst signal

write data line when it is time to write to the memory. The instruction register holds the values of the instruction being executed. The memory data register(MDR), stores the value read from memory. This will be written every clock cycle since the value only need be stored for one clock cycle. Thus, it does not need its own control signal and its enable can be attached to the clock. The register file, denoted by Registers in Figure 1, stores the 32 registers. Register A and B store the output of the register file for one clock cycle. The ALU does all the arithmetic computations. Its functionality is fully described by Table 3. The ALU OUT register holds the output of the ALU for one clock cycle. PCWrite Generate generates the enable signal to write to the PC. In the case of this design, it is described by the following equation: $G = PCWrite + PCWriteCondEq * zero + PCWriteCondNEq * \overline{zero}$.

2.1 Instruction Fetch, Instruction Decode, and Branch Target Computation

There are two steps that each instruction execution has in common. The first is the IF stage. Each instruction needs to be fetched from memory. To do this, we need to read the instruction from memory and store it into the instruction register. This is also the step where we must update the PC to $PC + 4$. So, with that in mind, we can define the signals needed to complete this step. First, IorD needs to be deasserted. This will select the PC as the address for memory. MemRead needs to be set, so that the instruction is fetched from memory. IRWrite needs to be asserted so that the instruction is stored after it is read. This is enough to actually fetch the instruction, but the PC still needs to be updated. Thus, we require ALUSrcA to be deasserted to select PC for port A of the ALU, and ALUSrcB to be set to ASB_FOUR to select 4 for the second port($PC + 4$). PCSource needs to be set to PS_PCINC inidcating that $PC + 4$ should be written to the PC. Finally, PCWrite needs to be asserted so that the value is written to the PC. One thing that should be noted is that all writes happen only when the clock is high, except for the IRWrite. This is a bug, however it does not affect the system in any way.

The next step is the instruction decode stage. In this stage, we do two things. First, we read the registers from the register file. This is done automatically by the instruction register. Since the ALU is not being used in this stage, its a good time to calculate the

MemToReg	
Signal value	Effect
MTR_ALUOUT	Register write data gets the output of the ALU OUT register
MTR_MDR	Register write data gets the output of the MDR register
MTR_PC	Register write data gets the PC

Table 6: Actions of MemToReg signal

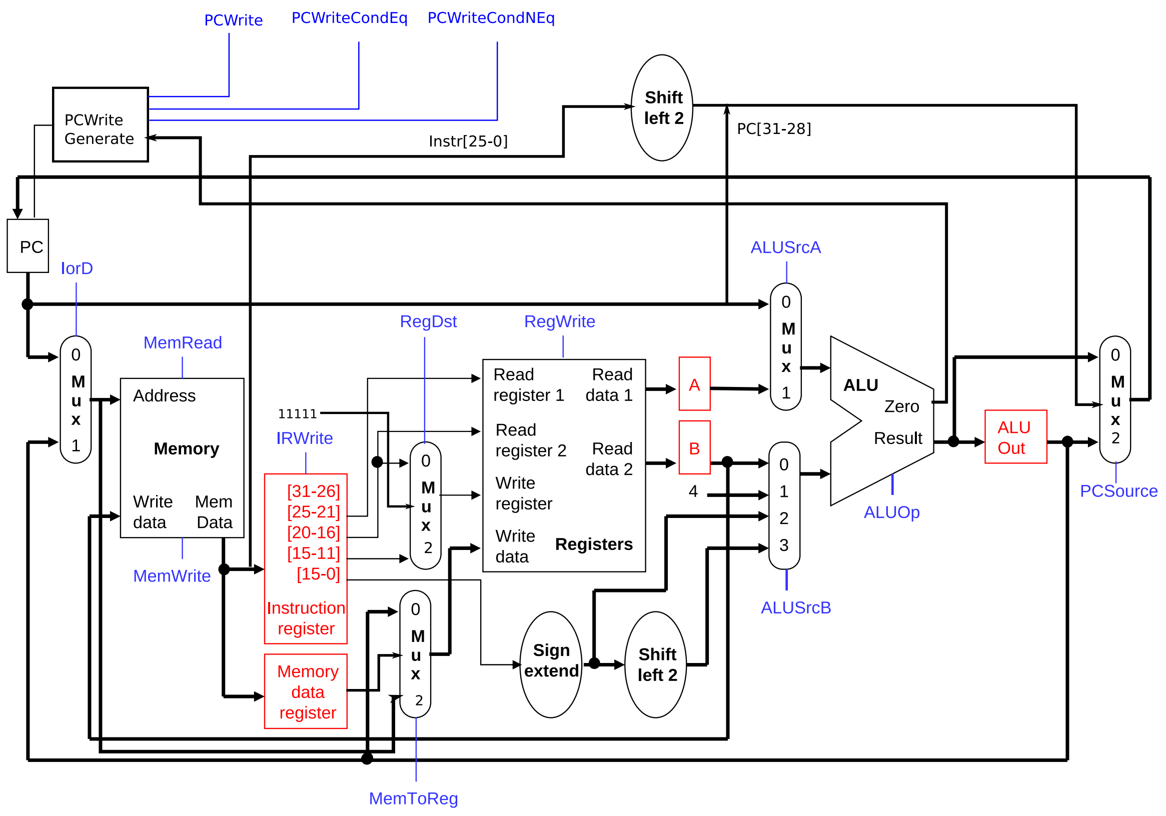


Figure 1: MIPS multicycle datapath

branch target address, regardless of if the instruction is a branch or not. This way, the target is ready incase it is a branch instruction. To do this, ALUSrcA is deasserted, indicating port A of the ALU gets the value of the PC. ALUSrcB is set to indicate the sign extended and shifted value of instruction[15-0]. These two will be added and stored in ALU OUT.

Figure 2 shows these two steps. One thing to note is that the first clock cycle is just a reset; it assigns values to all the signals so they are known. Thus, for this figure, and all future figures, the cycle begins in the second clock cycle. The important thing to note in this figure is that the PC is being assigned $PC + 4$ after executing the IF stage and IR has been assigned an instruction in that stage. In the next stage, register A and register B are assigned values that were read from the register file. The final clock cycle is bogus as the state machine was not updated.

2.2 LW/SW Instructions

LW and SW are the only two instructions allowed to access memory. They both have one step in common, and that is the memory address computation stage. In this step,

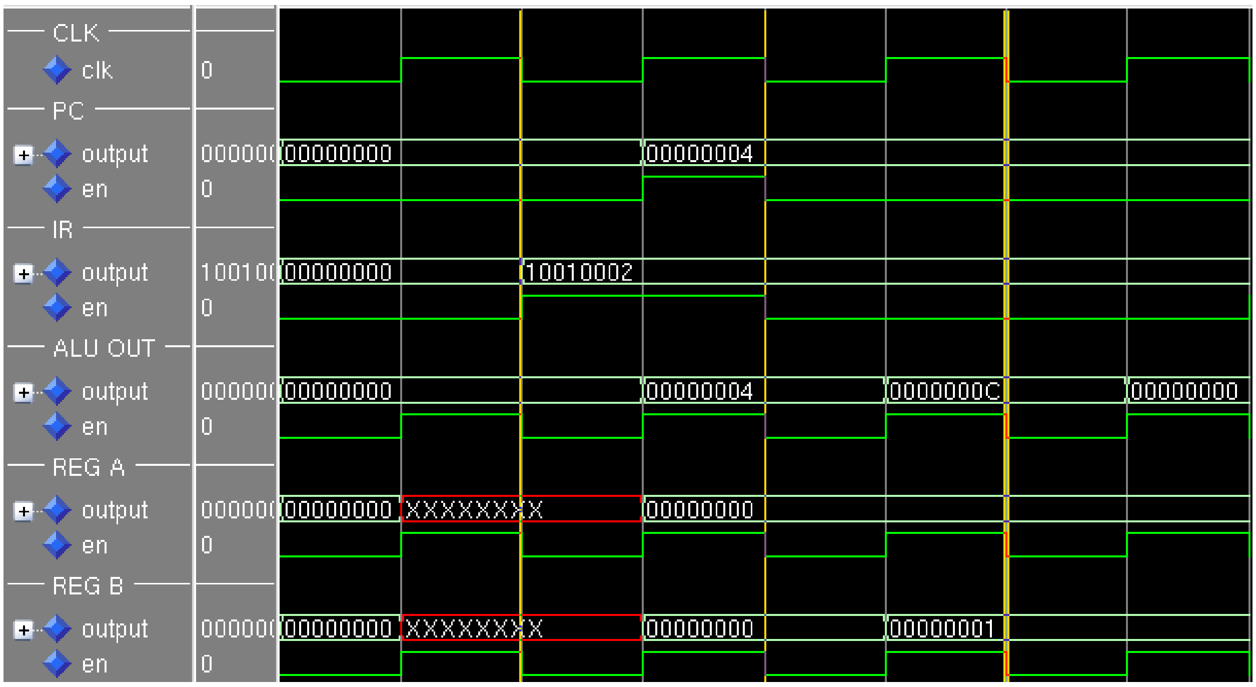


Figure 2: IF and ID stages. Both the first and last clock cycle have no meaning. The middle two are the one of interest.

ALUSrcA will be set to use register A. ALUSrcB will be set to use the bottom 16 bits of the instruction sign extended to 32 bits. These will be added by the ALU and used for the memory access.

Next is the memory access. The previously computed address is used here. Thus, in both the cases of LW and SW IorD is set to use the ALU OUT register. For LW, MemRead is asserted, and for SW, MemWrite is asserted.

At this point, SW is complete, but LW still has to write back. Thus, RegWrite is asserted, MemToReg is set to use the value stored in MDR, and RegDst is set to use RT.

Figure 3 shows the operation of LW. The first clock cycle is a reset. The second shows the IF stage. The third shows the ID stage. The fourth is where the memory address computation is occuring. Here, $0x00FF$ is being added to $R0$ as specified by the instruction. The ALU outputs FF as expected. Next, is the memory access. . . nothing happens here since the value of memory is kept constant for simplicity. Next, in the final clock cycle, the write back occurs, and $R1$ gets the value of the memory.

Similarly, Figure 4 shows the operation of SW. The first 4 cycles behave the same way. The final cycle is different in that it asserts a MemWrite signal and writes the data to memory.

2.3 R Type Instructions

This section shows the correct functionality of the R Type instructions. Only add and sll are described in detail. The waveforms for the remaining are at the end of the section. No further description is needed as only changing the ALU operation for the R Type instructions is done.

In general, R Type instructions consist of first performing the ALU operation specified, and writing that value back to the registers. For example, Figure 5 shows the waveform

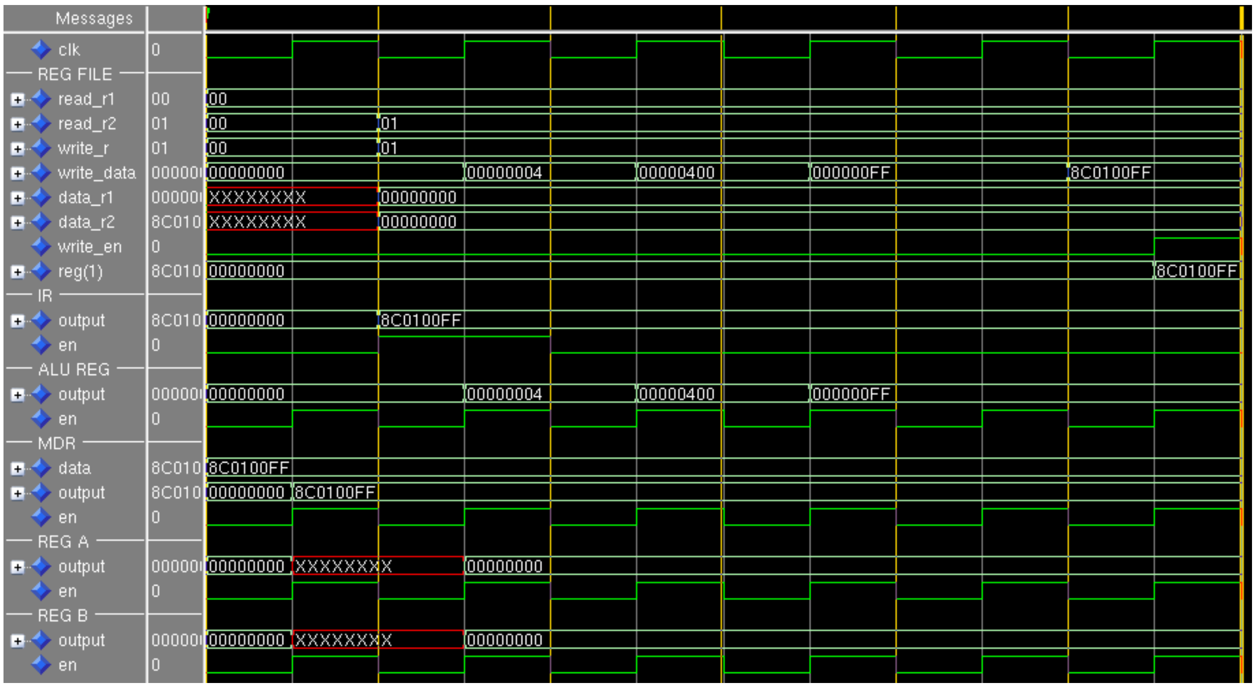


Figure 3: Shows the execution of a LW instruction

for an add instruction. First there is the reset cycle, then IF, and ID. Next is the execution stage. Here, the ALU takes the values read from the register file and performs and add on them. And the end of this cycle, write_data has the value of $R0 + R1$. In the next cycle, the write back cycle, $R2 <= R0 + R1$ as expected.

The SLL is similar, except instead of adding 2 operands, it takes the B operand to the ALU and shifts it by an amount specified in the instruction. This is shown in Figure **??**. The first clock cycle is the reset, followed by the IF, followed by ID, followed by the execution. In the execution, ALUSrcA is irrelevant. Operand B(R1) is '1', which will be left shifted by the amount of 2. Thus, the expected value is 4, and that is the value that is written back in the final clock cycle.

The following are the figures for SUB, AND, OR, and SLT. Whats important to look at in these instructions are the operands and the final write back values. With that, it can be seen that each functions correctly.

2.4 Immediate Instruction

Only one immediate instruction required testing, and the was ADDI. Its very similar to ADD, except that ALUSrcB is set to use the sign extended lower 16 bits of the instruction. Its waveform is shown in Figure 11. The first cycle is reset, the second is instruction fetch, third is instruction decode, fourth is execution. The lower 16 bits of this instruction are 8, which is being added to $R0$. The final clock cycle shows that 8 is written back as expected.

2.5 Branch and Jump

This section shows the correct functionality of BNE and JAL. BEQ and JMP are shown at the end of this section for completeness, however they are not annotated as the instructions are very similar to BNE and JAL.

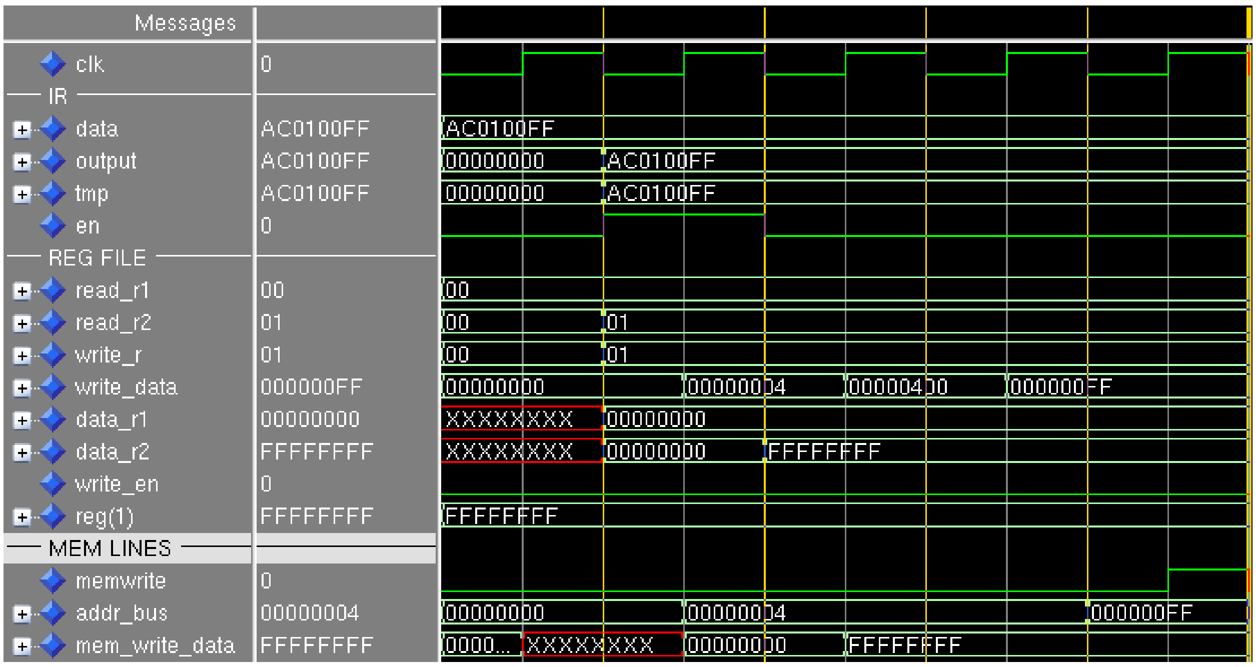


Figure 4: Shows the execution of a SW instruction

The correct functionality for BNE is shown in Figure 12. For branch instructions, the target address is available after the ID stage. Thus, as soon as the appropriate signals have progagated from the ALU, we can branch. For BNE, the control unit will set ALUSrcA to register A. ALUSrcB will be set to allow register B. These two will be compared in the ALU during the execution stage. Along with those signals, BNE needs to assert the PCWriteCondNEq signal to indicate a branch when not equal. The branch occurs in the execution stage, as that is when PC is written with its new value. In Figure 12, the lower 16 bits are 2. Sign extended and shifted left by two makes 8. Thus, $PC + 4 + 8 = C$, as the original PC started off at 0. This is the value that is written to the PC as can be seen.

Jumping is similar to a branch, however it uses instruction[26-0] shifted left by 2 for the lower 28 bits. The higher bits are determined by PC[31-28]. JAL, shown in Figure 13, does more than just jump. It also writes the value of the PC to $R31$. The instruction if Figure 13 stores the value of PC as expected into $R31$ in the final clock cycle. The lower 26 bits of the instruction are 8, thus $0x20$ is the expected jump value, the same value that PC gets in the final clock cycle.

The following figures(13 and 15 show similar instructions J and BEQ.

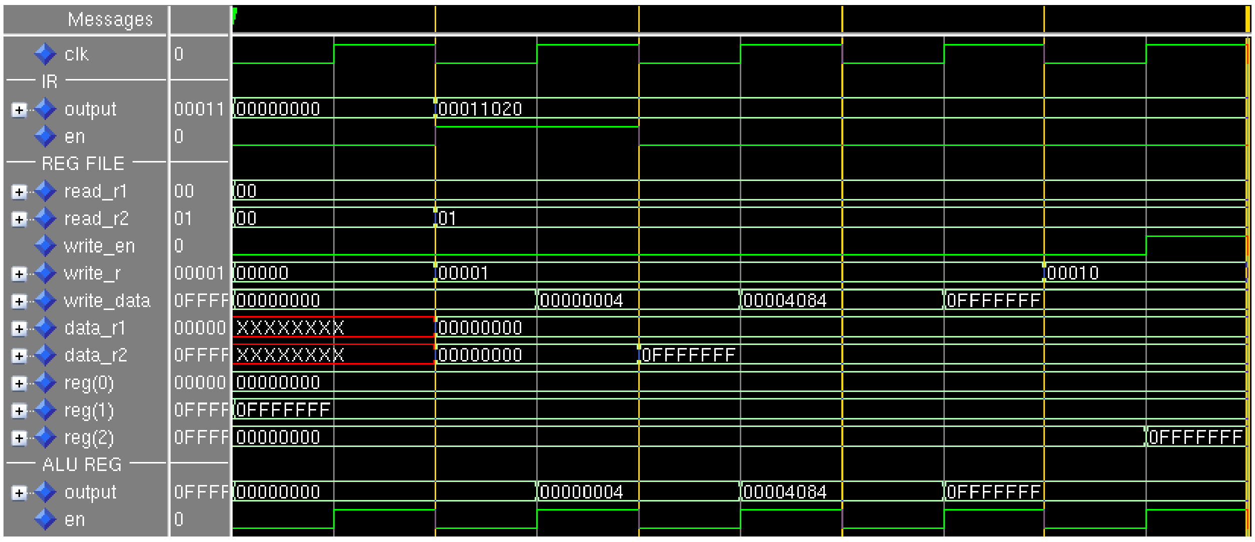

Messages
clk 0
IR
output 00011 00000000 00011020
en 0
REG FILE
read_r1 00 00
read_r2 01 00 01
write_en 0
write_r 00001 00000 00001 00010
write_data 0FFFF 00000000 00000004 00004084 0FFFFFFF
data_r1 00000 XXXXXXXX 00000000
data_r2 0FFFF XXXXXXXX 00000000 0FFFFFFF
reg(0) 00000 00000000
reg(1) 0FFFF 0FFFFFFF
reg(2) 0FFFF 00000000 0FFFFFFF
ALU REG
output 0FFFF 00000000 00000004 00004084 0FFFFFFF
en 0

Figure 5: Shows the execution of a add instruction

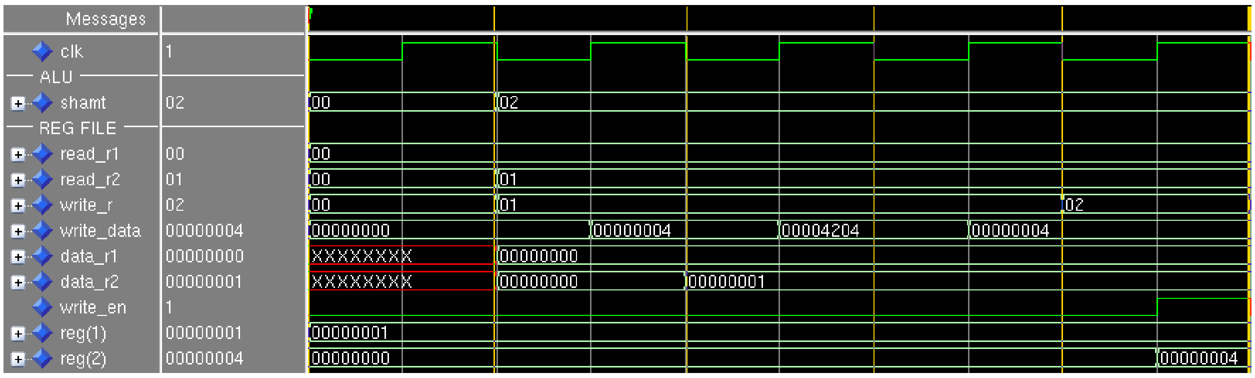

Messages
clk 1
ALU
shamt 02 00 02
REG FILE
read_r1 00 00
read_r2 01 00 01
write_r 02 00 01 02
write_data 00000004 00000000 00000004 00004204 00000004
data_r1 00000000 XXXXXXXX 00000000
data_r2 00000001 XXXXXXXX 00000000 00000001
write_en 1
reg(1) 00000001 00000001
reg(2) 00000004 00000000 00000004

Figure 6: Shows the execution of a SLL instruction

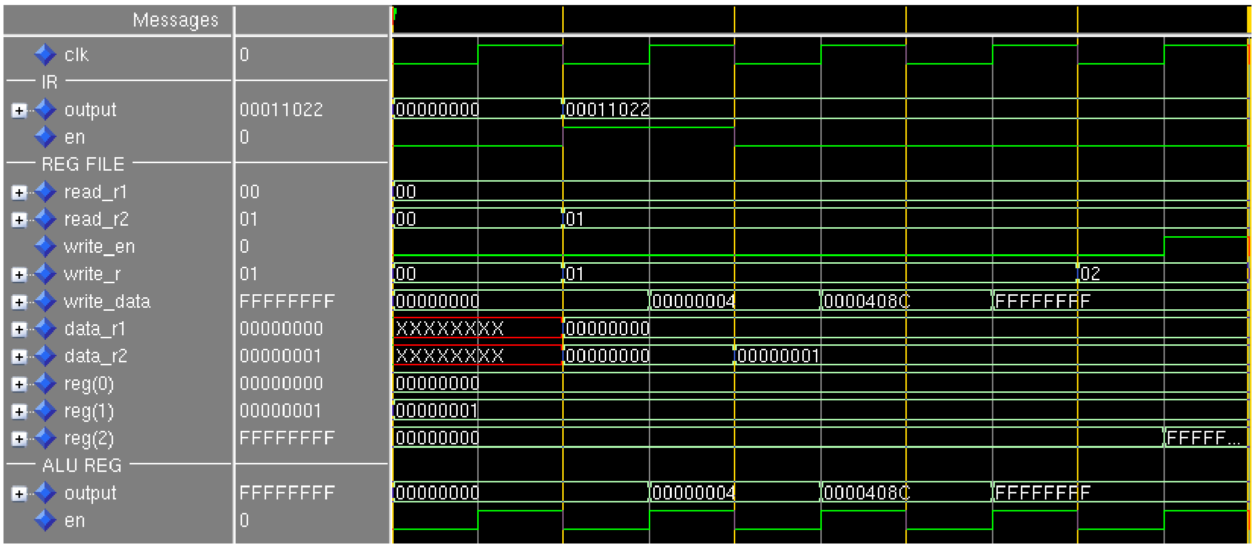
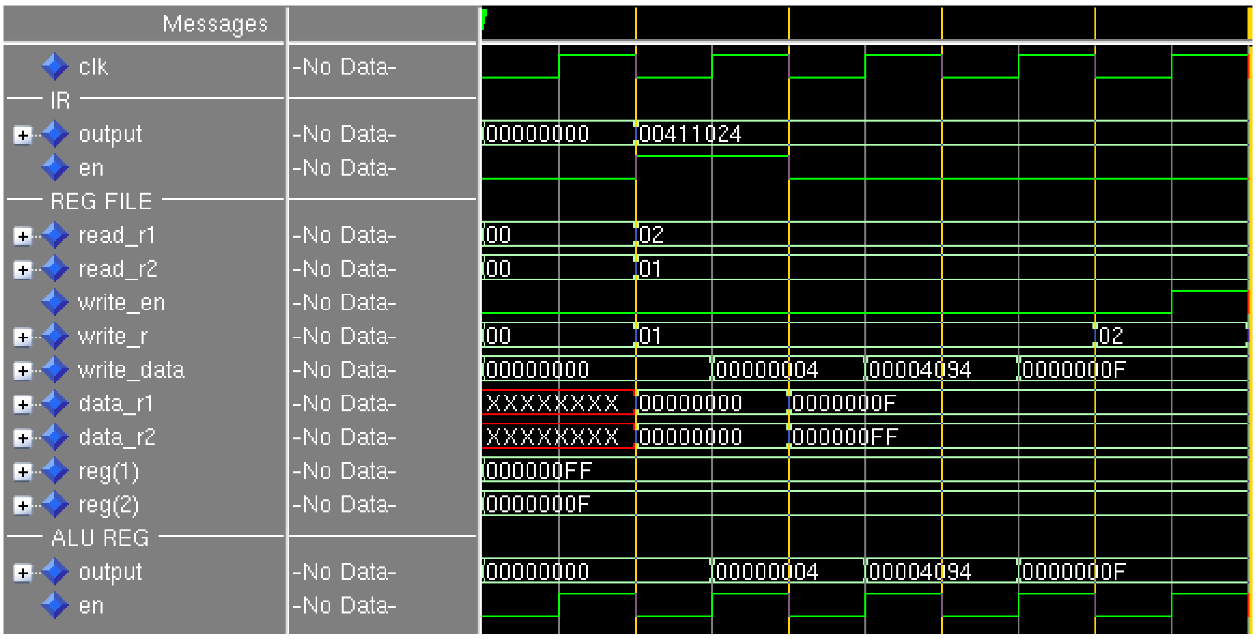
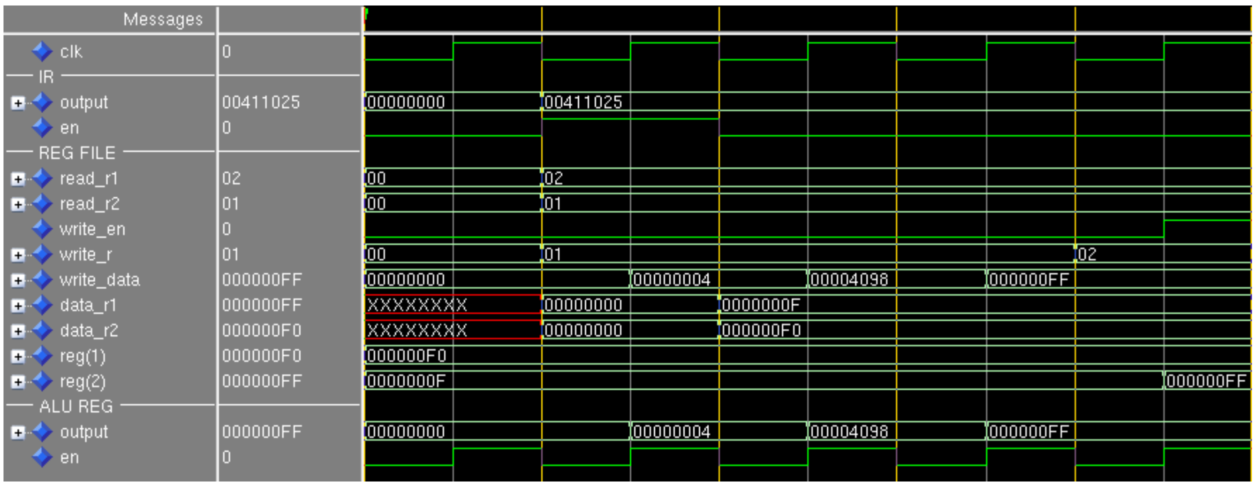

Messages
clk 0
IR
output 00011022 00000000 00011022
en 0
REG FILE
read_r1 00 00
read_r2 01 00 01
write_en 0
write_r 01 00 01 02
write_data FFFFFFFF 00000000 00000004 0000408C FFFFFFFF
data_r1 00000000 XXXXXXXX 00000000
data_r2 00000001 XXXXXXXX 00000000 00000001
reg(0) 00000000 00000000
reg(1) 00000001 00000001
reg(2) FFFFFFFF 00000000 FFFFF...
ALU REG
output FFFFFFFF 00000000 00000004 0000408C FFFFFFFF
en 0

Figure 7: Shows the execution of a SUB instruction



Messages
clk -No Data-
IR
output -No Data- 00000000 00411024
en -No Data-
REG FILE
read_r1 -No Data- 00 02
read_r2 -No Data- 00 01
write_en -No Data-
write_r -No Data- 00 01 02
write_data -No Data- 00000000 00000004 00004094 0000000F
data_r1 -No Data- XXXXXXXX 00000000 0000000F
data_r2 -No Data- XXXXXXXX 00000000 000000FF
reg(1) -No Data- 000000FF
reg(2) -No Data- 0000000F
ALU REG
output -No Data- 00000000 00000004 00004094 0000000F
en -No Data-

Figure 8: Shows the execution of a AND instruction



Messages
clk 0
IR
output 00411025 00000000 00411025
en 0
REG FILE
read_r1 02 00 02
read_r2 01 00 01
write_en 0
write_r 01 00 01 02
write_data 000000FF 00000000 00000004 00004098 000000FF
data_r1 000000FF XXXXXXXX 00000000 0000000F
data_r2 000000F0 XXXXXXXX 00000000 000000F0
reg(1) 000000F0 000000F0
reg(2) 000000FF 0000000F 000000FF
ALU REG
output 000000FF 00000000 00000004 00004098 000000FF
en 0

Figure 9: Shows the execution of a OR instruction

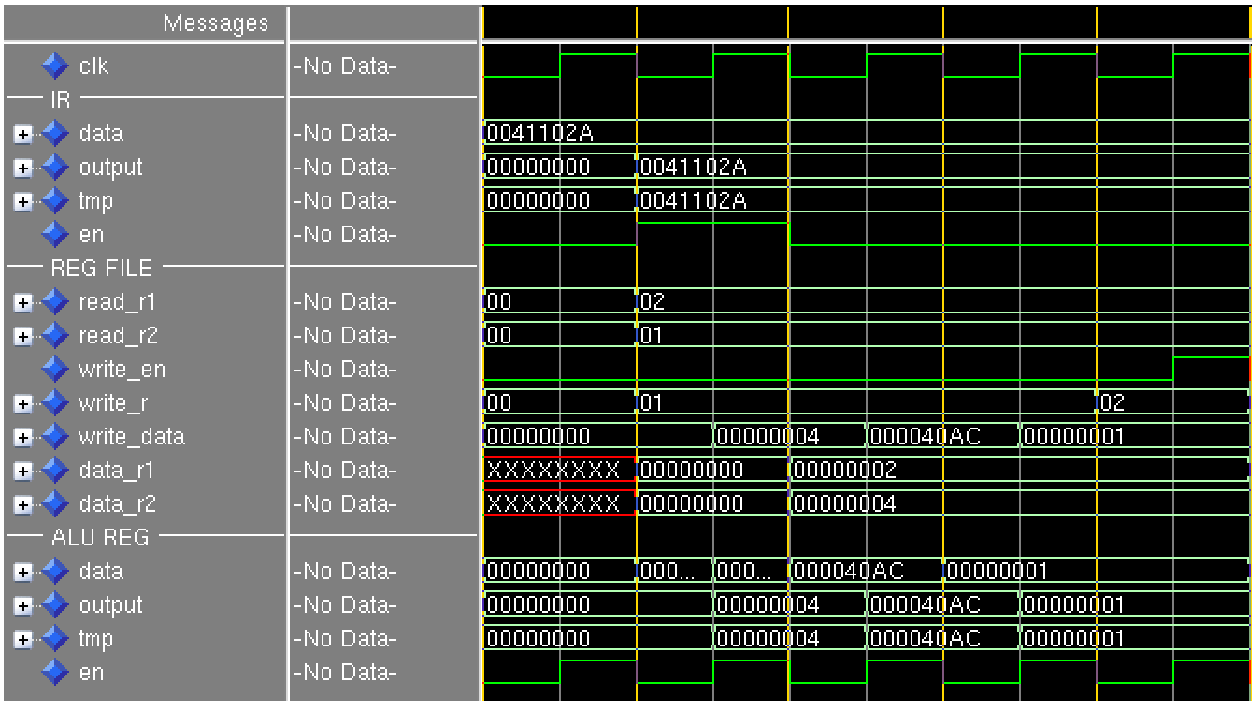

Messages
clk -No Data-
IR
data -No Data- 0041102A
output -No Data- 00000000 0041102A
tmp -No Data- 00000000 0041102A
en -No Data-
REG FILE
read_r1 -No Data- 00 02
read_r2 -No Data- 00 01
write_en -No Data-
write_r -No Data- 00 01 02
write_data -No Data- 00000000 00000004 000040AC 00000001
data_r1 -No Data- XXXXXXXX 00000000 00000002
data_r2 -No Data- XXXXXXXX 00000000 00000004
ALU REG
data -No Data- 00000000 000... 000... 000040AC 00000001
output -No Data- 00000000 00000004 000040AC 00000001
tmp -No Data- 00000000 00000004 000040AC 00000001
en -No Data-

Figure 10: Shows the execution of a SLT instruction

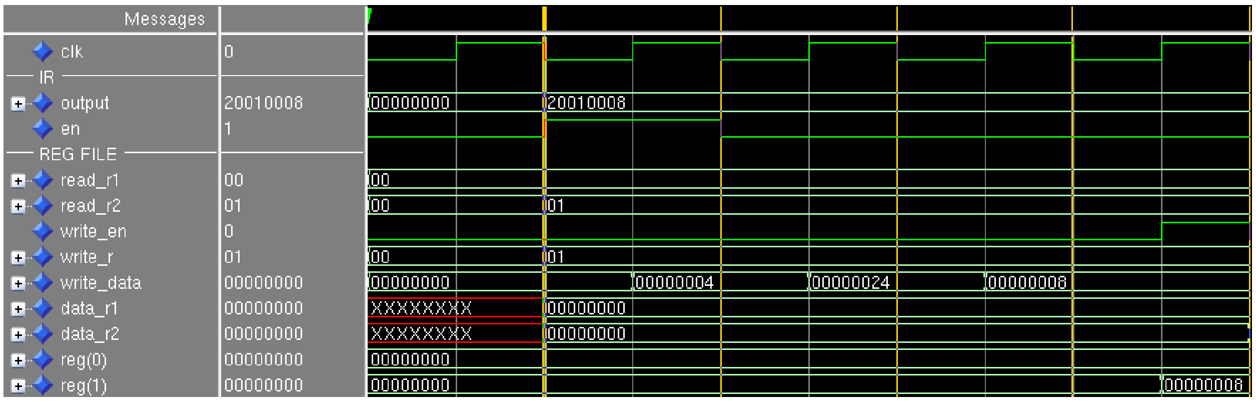

Messages
clk 0
IR
output 20010008 00000000 20010008
en 1
REG FILE
read_r1 00 00
read_r2 01 00 01
write_en 0
write_r 01 00 01
write_data 00000000 00000000 00000004 00000024 00000008
data_r1 00000000 XXXXXXXX 00000000
data_r2 00000000 XXXXXXXX 00000000
reg(0) 00000000 00000000
reg(1) 00000000 00000000 00000008

Figure 11: Shows the execution of a ADDI instruction

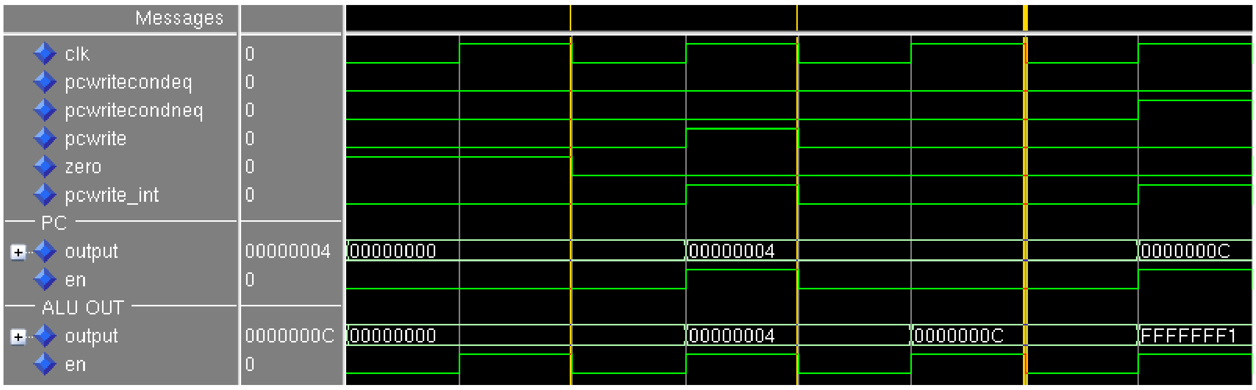

Messages
clk 0
pcwritecondeq 0
pcwritecondneq 0
pcwrite 0
zero 0
pcwrite_int 0
PC
output 00000004 00000000 00000004 0000000C
en 0
ALU OUT
output 0000000C 00000000 00000004 0000000C FFFFFFF1
en 0

Figure 12: Shows the execution of a BNE instruction

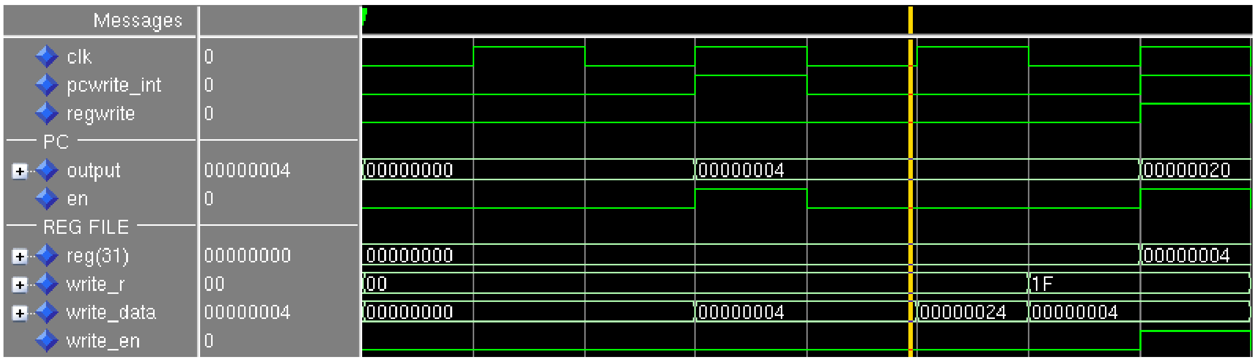


Figure 13: Shows the execution of a JAL instruction

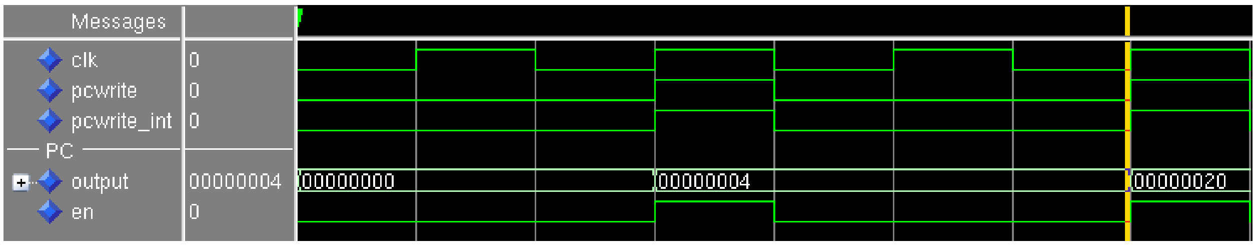


Figure 14: Shows the execution of a J instruction

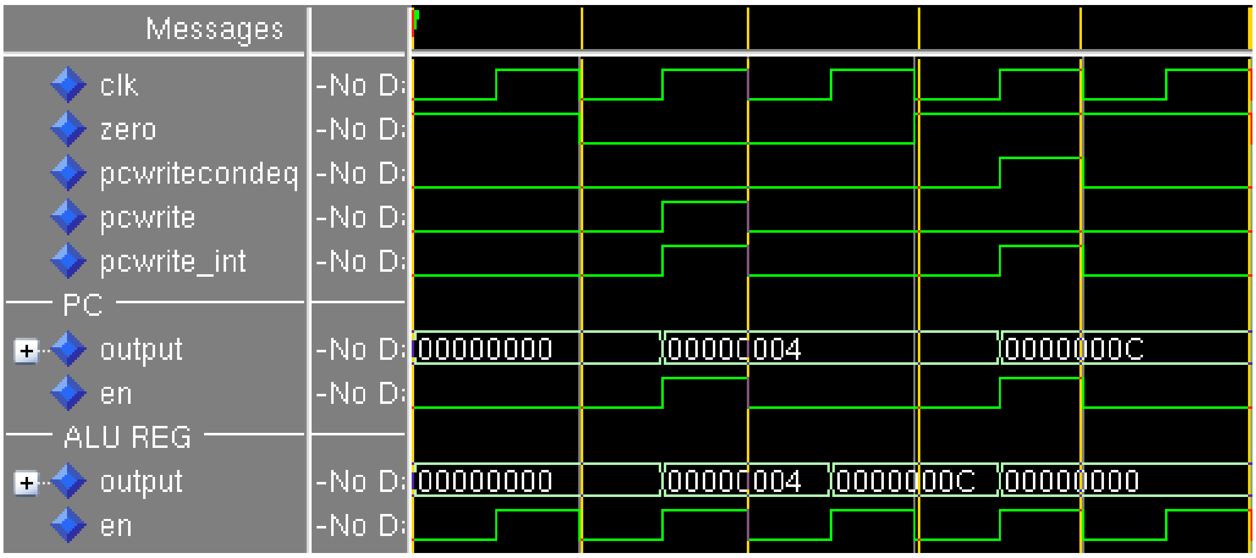


Figure 15: Shows the execution of a BEQ instruction