CmpE344.02 Computer Organization
Project 1 BU2020 Report

Ömer Cihan Benzer 2017400048
Yunus Emre Topal 2018400075

Introduction

In this project, we designed a 16 bit-RISC processor named BU2020 processor, using iverilog.

The Processor

Stages:

BU2020 processor has 5 stages:

1. Instruction fetch, IF, handles the program counter and fetches the next instruction.
2. Instruction decode, ID, both handles the control unit and puts the necessary data in order, either from registers or as immediate value.
3. Execution, EXE, using the sorted data and the ALU, gets the necessary data as output. EXE stage also calculates the next PC for a possible Jump instruction.
4. Memory, MEM, if needed (controlled via the control unit bits), fetches data from the memory or sends the output into the memory from/to the given address.
5. Write Back, WB, if needed(controlled via the control unit bits), sends the final data to the register block, into the given register.

Control Unit:

The Control unit which is in the “ID” stage handles the controller/selector bits of the whole processor. Overall there are 12 bits which 5 bit goes to EX, 5 bit goes to MEM and 2 bit goes to WB stages:

- 5 bit to EX, 3 bit as the ALU instruction and 2 bit as the selector of 4-to-1 MUX that selects the 2nd ALU input.
- 5 bits to MEM, 1 bit for each for doubleRead and doubleWrite, which are responsible for the indirect instructions. 1 Bit for read/write instruction and 1 bit each for BNE and JMP instructions.
- 2 bits to WB, 1 bit as the selector of 2-1 MUX that selects the writeback bus, which is either memory or ALU output. The final bit is the RegWrite bit which is if 1, the register block writes the given data to the given register.

Note that these control bits have the required registers to be held until the pipeline reaches the assigned stage.

Register Block:
The register block has 3 data buses: 2 for immediate read and 1 for sequential write. There are 3 address bits that resemble 8 registers, the special 000 register is also known as BA register.

Arithmetic Logic Unit
ALU is a module that takes 2 inputs and a 3-bit instruction, which gives 2 16-bit outputs: the status register and the output.

- Status register is a register in which only the first 4 bits are being used. These bits carry the flag, in order: Zero flag, Negative flag, Carry flag and the Overflow flag.
- Output is the data that is wanted, this can be the Sum, Subtraction, Bitwise AND etc.. depending on the ALU instruction.

Memory
The memory of the BU2020 processor is a memory segment with a size of 4x1024 bytes, the first called the instruction memory. However, if needed, the other 3 data memories can also be used as instruction addresses, the PC is capable of reading data from those memories as well.
There are 3 busses, 2 immediate output and 1 sequential input. Since the memory holds both the instructions and the necessary data, we need 2 output buses that can work simultaneously. The input bus is loaded with the necessary data and if write_mode bit is 1, the data is written into the memory to the specified address.

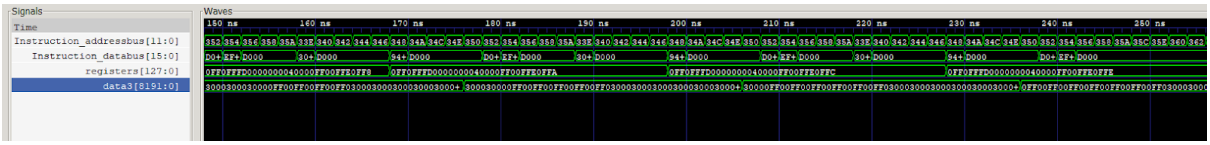
Note that there is also the doubleRead and doubleWrite bits, which are responsible for the indirect read/write instructions. That is, as an example, if doubleRead is 1, the memory reads the data of the given address, then reads from the address that is in that data, and gives the 2nd read data as the output.

Source Code
Our source code is available on GitHub. There are a total of 7 files and a batch file that eases the run.
Link to the GitHub Repository

Evaluation of the Processor
According to iverilog standards, if we remove the “initial” statements, which is possible, our source code is synthesizable for ASIC; however, we couldn’t manage to open a program that can synthesize our code neither for FPGA nor ASIC. Therefore, even if our code is synthesizable, we can only theoretically evaluate our processor and we couldn’t benchmark it.
Unfortunately, in our processor, there is no hazard control which means the compiler should add stalls between potential hazards manually. However, if we want to ensure there won’t be any problem caused by a data hazard, the worst case

would be that the processor works at 5CPI, when we send each instruction at every 5 cycle.
We still need an average, hence we wrote a small script that iterates over the memory and writes data N times, while the specified register is not equal to 0xFFE.

Here is the final 4 iterations of the script, note that the data segment is getting filled with 0x0FF0 at each iteration:



From this script, we can say that our processor works at 3.6 CPI on average.

Instructions

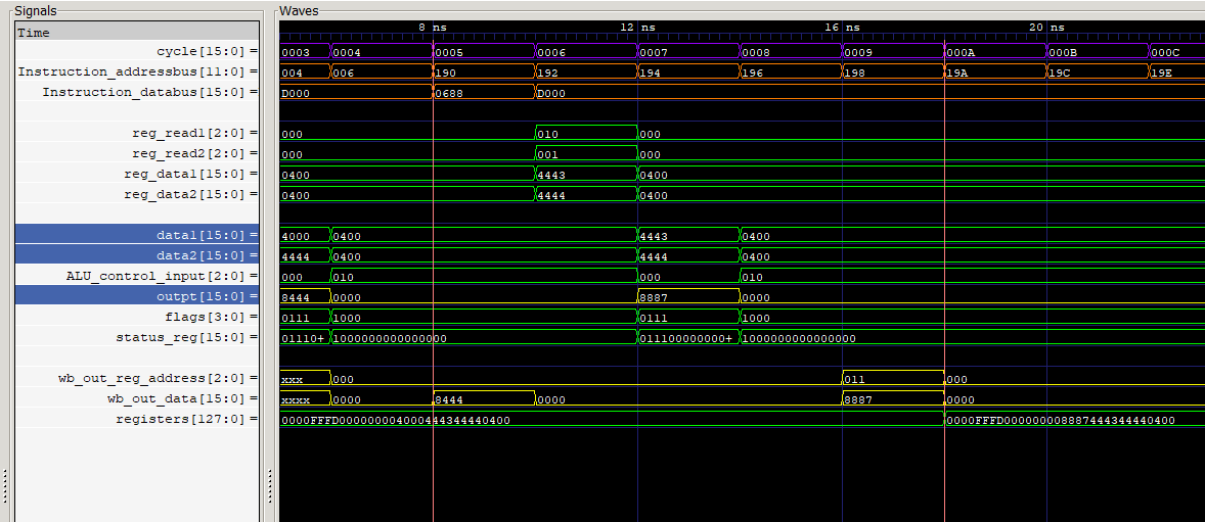
The project had 16 instructions that was given, we managed to handle all but one, which is the opcode 0001:

“Add d0,d1, 25 -> d0 = d1 + memory(BA+25*2)”

We couldn’t do the instruction as our processor handles the Arithmetic Logic operations before accessing the memory, which makes the operation impossible for us in one cycle.
Below are the 15 instructions on TestBench:

ADD:

Example instruction => 0688h = 0000 011 010 001 XXX = Add R3, R2, R1



Cycle 5: Instruction Fetch

Cycle 6: Instruction Decode. Rs1 = R2 = 4443h, Rs2 = R1 = 4444h, Rd = R3.

Cycle 7: Execution. R2 + R1 = 8887h.

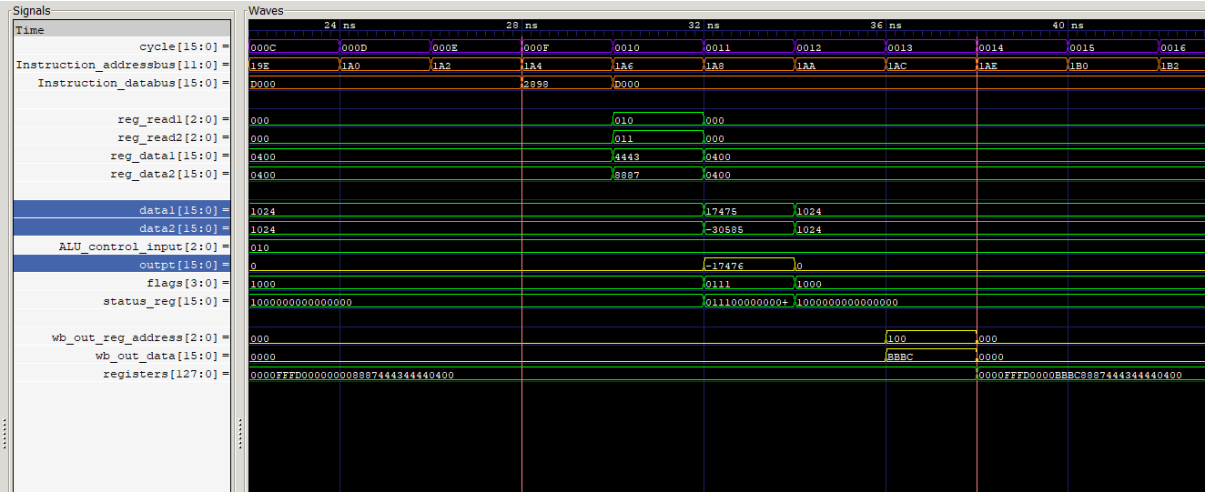
Cycle 8: This instruction does not have a memory operation.

Cycle 9: Write back to R3 = 8887h.

You can see values of registers on the Cycle 10.

SUB:

Example Instruction => 2898h = 0010 100 010 011 XXX = Sub R4, R2, R3



Cycle 15: Instruction Fetch

Cycle 16: Instruction Decode. Rs1 = R2 = 4443h, Rs2 = R3 = 8887h, Rd = R4

Cycle 17: Execution. R2 – R3 = BBBCh.

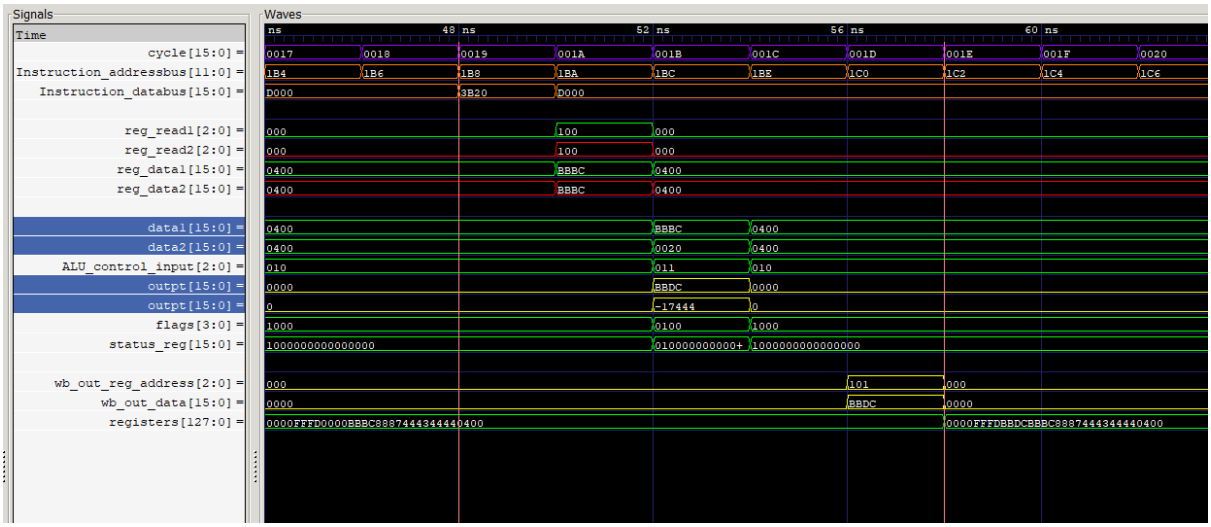
Cycle 18: This instruction does not have a memory operation.

Cycle 19: Write back to R4 = BBBCh.

You can see values of registers on the Cycle 20.

ADDI:

Example Instruction => 3B20h = 0011 101 100 100000 = Addi R5, R4, 32



Cycle 25: Instruction Fetch

Cycle 26: Instruction Decode. Rs1 = R4 = BBBCh, immediate = 0020h, Rd = R5.

Cycle 27: Execution. R4 + 0020h = BBDCh.

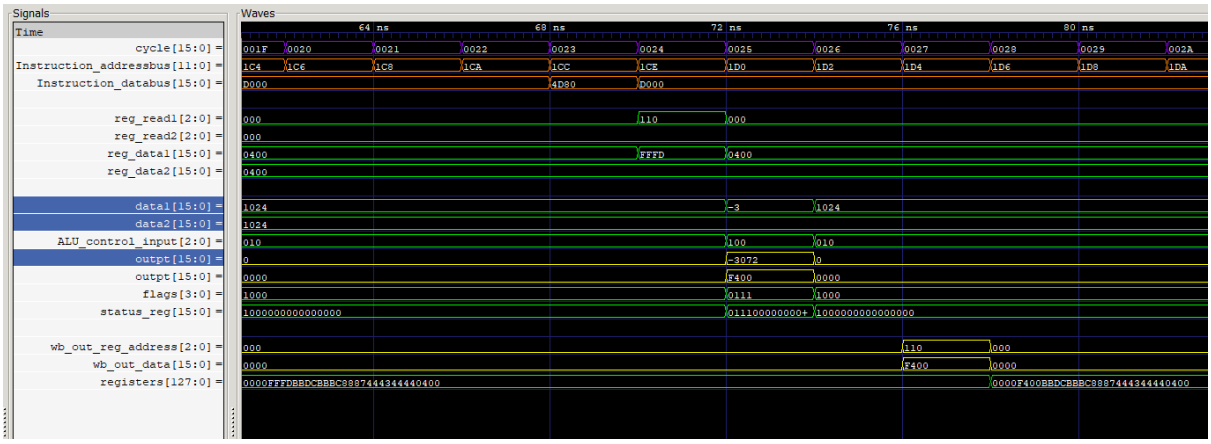
Cycle 28: This instruction does not have a memory operation.

Cycle 29: Write back to R5 = BBDCh.

You can see values of registers on the Cycle 30.

MUL:

Example Instruction => 4D80h = 0100 110 110 000 XXX = Mul R6, R6, R0



Cycle 35: Instruction Fetch

Cycle 36: Instruction Decode. Rs1 = R6 = FFFDh, Rs2 = R0 = 0400h, Rd = R6.

Cycle 37: Execution. R6 x R0 = F400h.

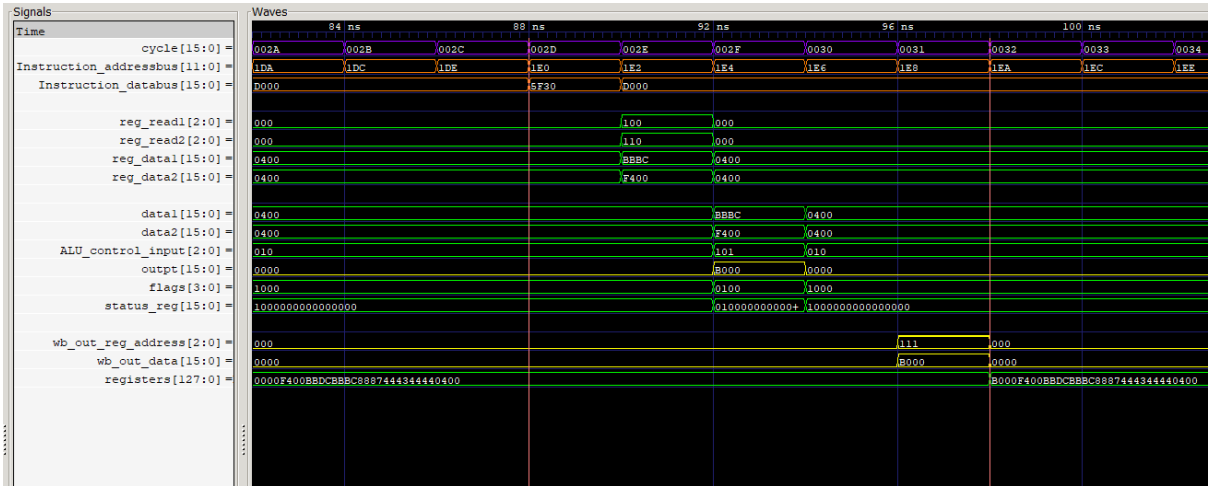
Cycle 38: This instruction does not have a memory operation.

Cycle 39: Write back to R6 = F400h.

You can see values of registers on the Cycle 40.

AND:

Example Instruction => 5F30h = 0101 111 100 110 XXX = And R7, R4, R6



Cycle 45: Instruction Fetch

Cycle 46: Instruction Decode. Rs1 = R4 = BBBCh, Rs2 = R6 = F400h, Rd = R7.

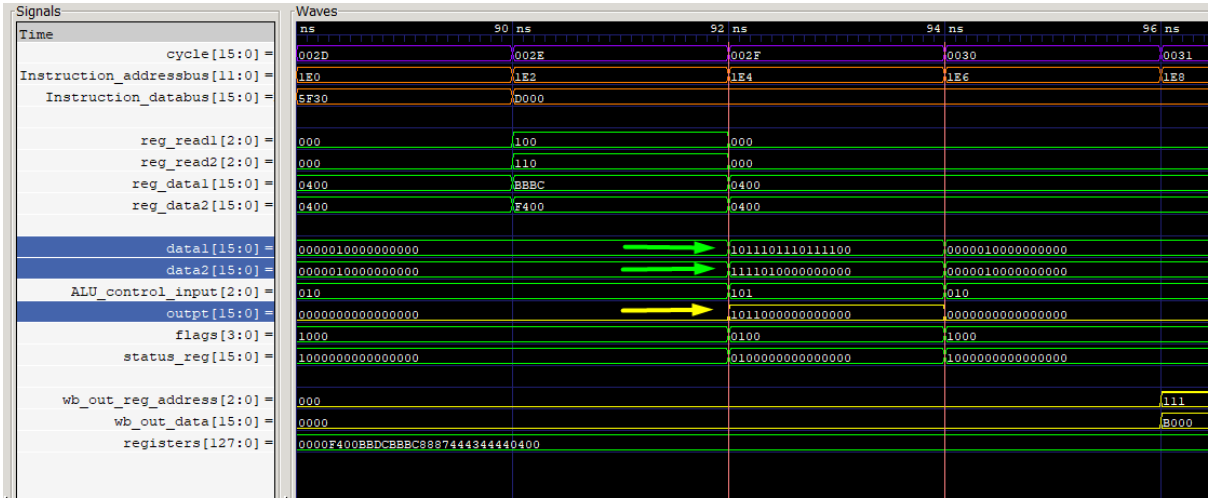
Cycle 47: Execution. R4 & R6 = B000h.

Cycle 48: This instruction does not have a memory operation.

Cycle 49: Write back to R7 = B000h.

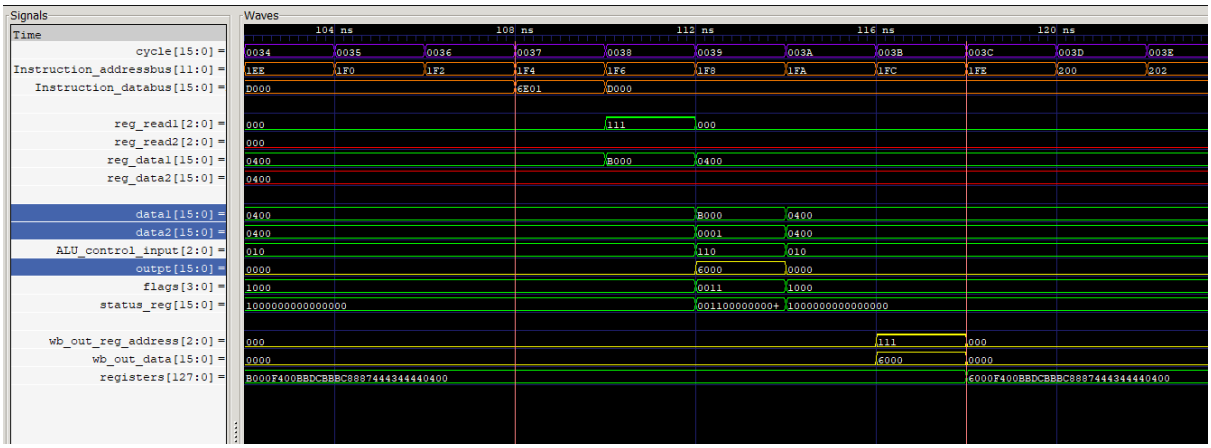
You can see values of registers on the Cycle 50.

Binary version of the values:



SLL:

Example Instruction => 6E01h = 0110 111 000000001 = Sll R7, 1



Cycle 55: Instruction Fetch

Cycle 56: Instruction Decode. Rs1 = R7 = B000h, immediate = 0001h, Rd = R7.

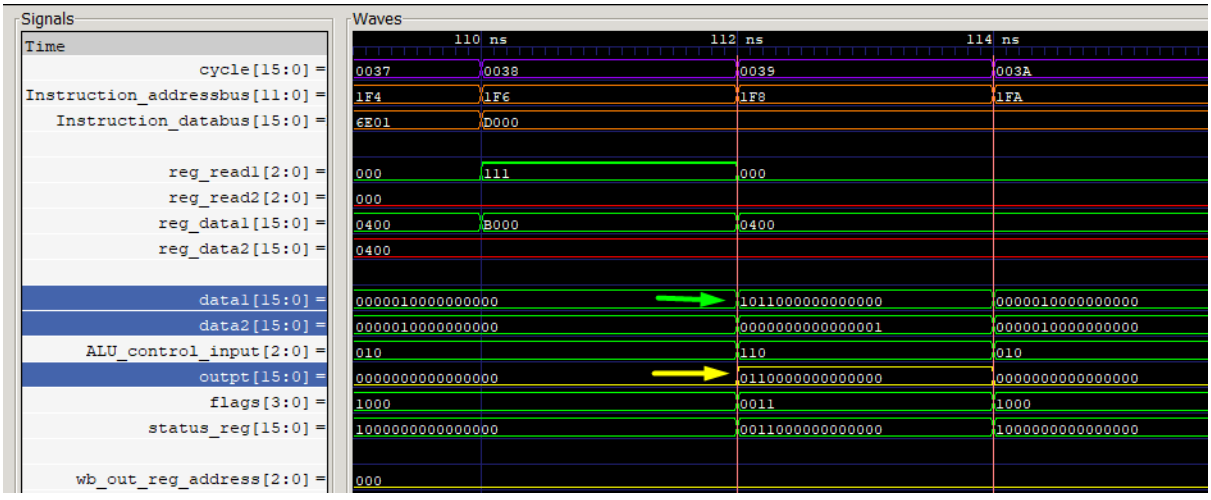
Cycle 57: Execution. R7 << 1 = 6000h.

Cycle 58: This instruction does not have a memory operation.

Cycle 59: Write back to R7 = 6000h.

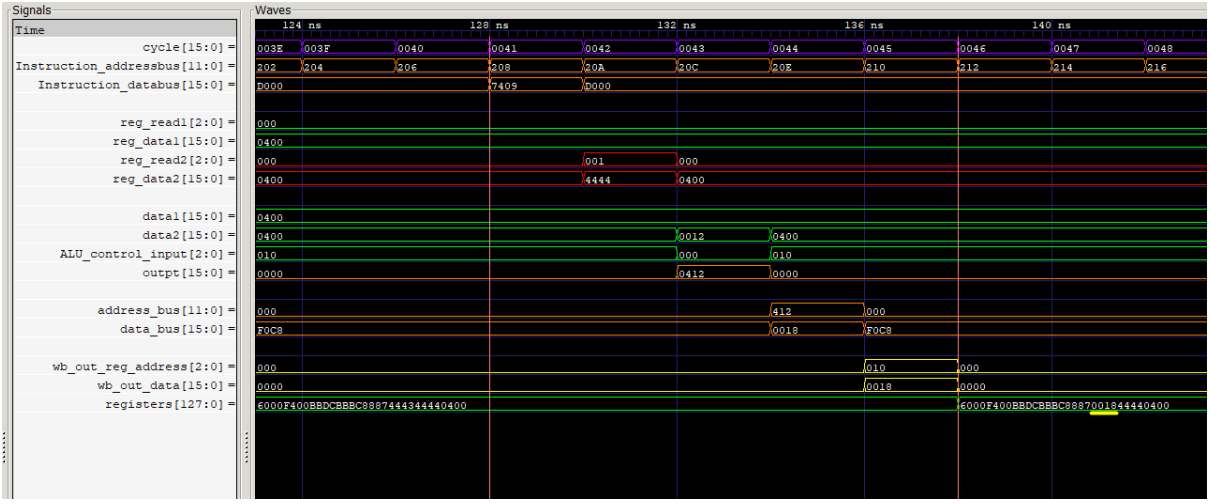
You can see values of registers on the Cycle 60.

Binary version of the values:



LW:

Example Instruction => 7409h = 0111 010 000001001 = Lw R2, 9



Cycle 65: Instruction Fetch

Cycle 66: Instruction Decode. immediate = 0007h, Rd = R4.

Cycle 67: Execution. BA + 9 x 2 = 0400h + 0012h = 0412h.

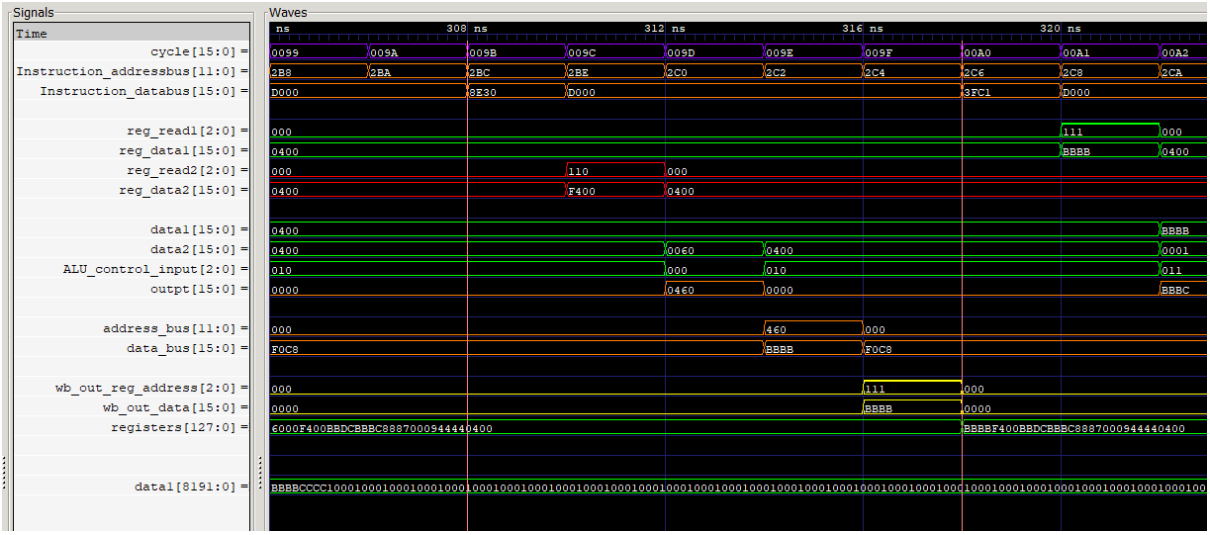
Cycle 68: Memory operation. R4 = memory[0412h] = 0018h.

Cycle 69: Write back to R4 = 0018h.

You can see values of registers on the Cycle 70.

LWI:

Example Instruction => 8E30 = 1000 111 000110000 = Lwi R7, 48



Cycle 155: Instruction Fetch

Cycle 156: Instruction Decode. immediate = 0030h, Rd = R7.

Cycle 157: Execution. BA + 48 x 2 = 0400h + 0060h = 0460h.

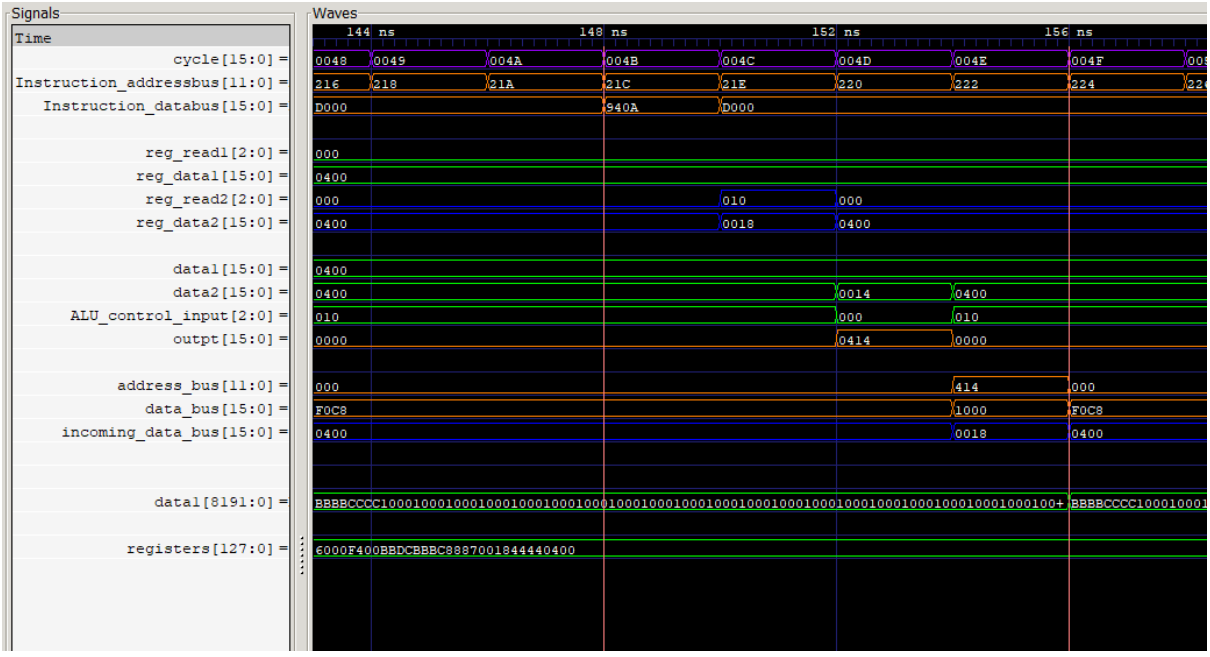
Cycle 158: Memory operation. R7 = memory[memory[0460h]] = BBBBh.

Cycle 159: Write back to R7 =BBBBh.

You can see values of registers on the Cycle 160.

SW:

Example Instruction => 940Ah = 1001 010 000001010 = Sw R2, 10



Cycle 75: Instruction Fetch

Cycle 76: Instruction Decode. immediate = 000Ah, Rd = R2.

Cycle 77: Execution. BA + 10 x 2 = 0400h + 0014h = 0414h.

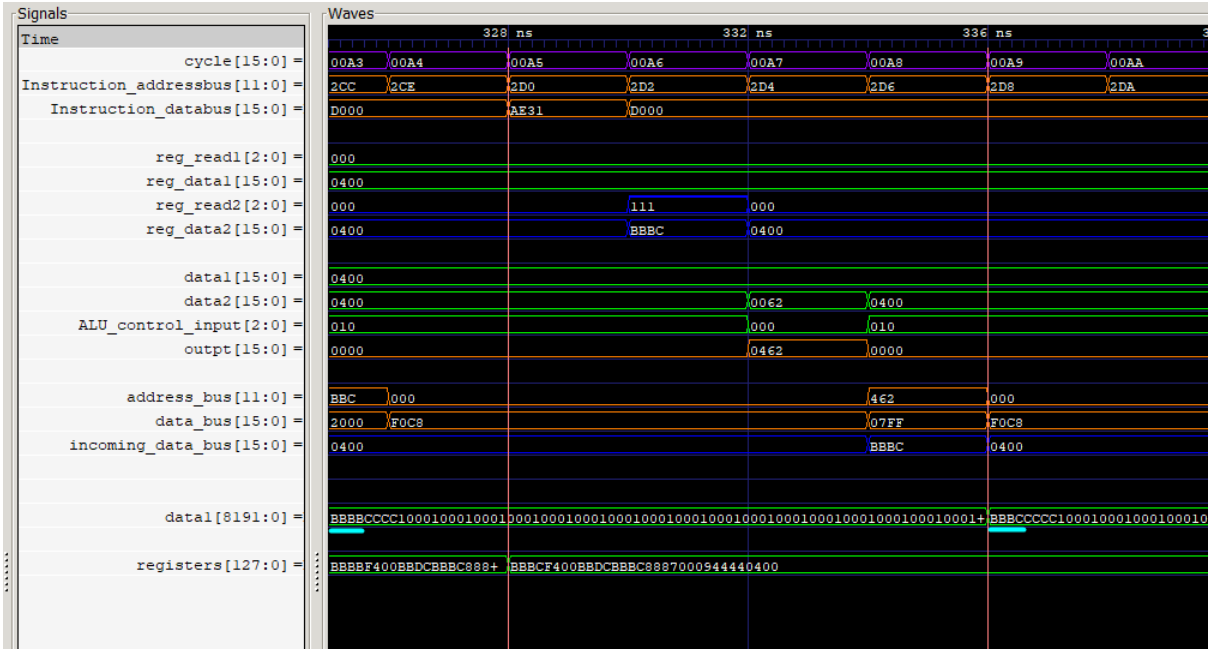
Cycle 78: Memory operation. memory[0414h] = R2 = 0018h.

Cycle 79: This intruction does not have write back operation.

You can see values of registers on the Cycle 80.

SWI:

Example Instruction => AE31h = 1010 111 000110001 = Swi R7, 49



Cycle 165: Instruction Fetch

Cycle 166: Instruction Decode. immediate = 0031h, Rd = R7.

Cycle 167: Execution. BA + 49 x 2 = 0400h + 0062h = 0462h.

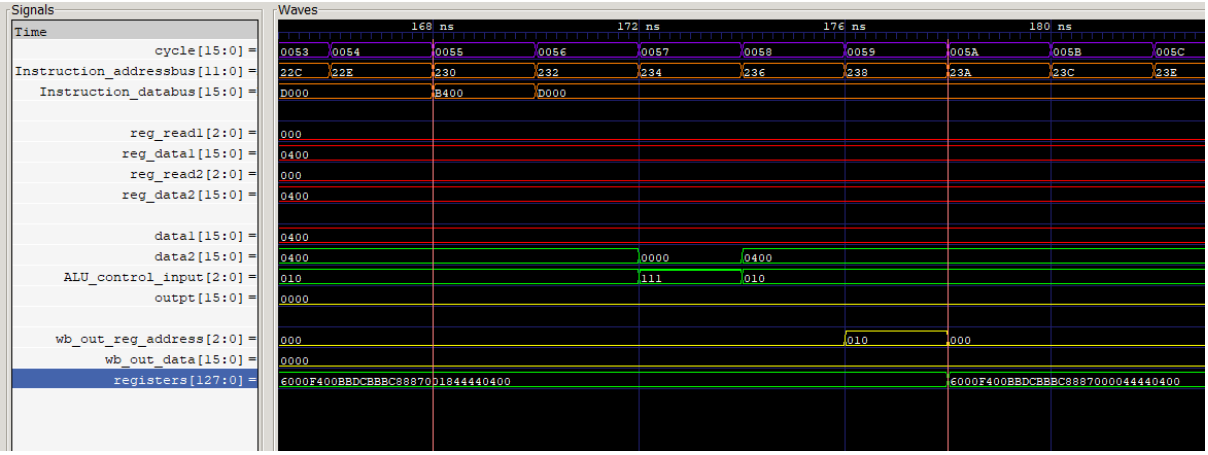
Cycle 168: Memory operation. Memory[memory[0460h]] = R7 = BBBCh.

Cycle 169: This intruction does not have write back operation.

You can see values of registers on the Cycle 170.

CLR:

Example Instruction => B400 = 1011 010 XXXXXXXXX = Clr R2



Cycle 85: Instruction Fetch

Cycle 86: Instruction Decode. immediate = 0031h, Rd = R2.

Cycle 87: Execution. Outpt wire is 0000h.

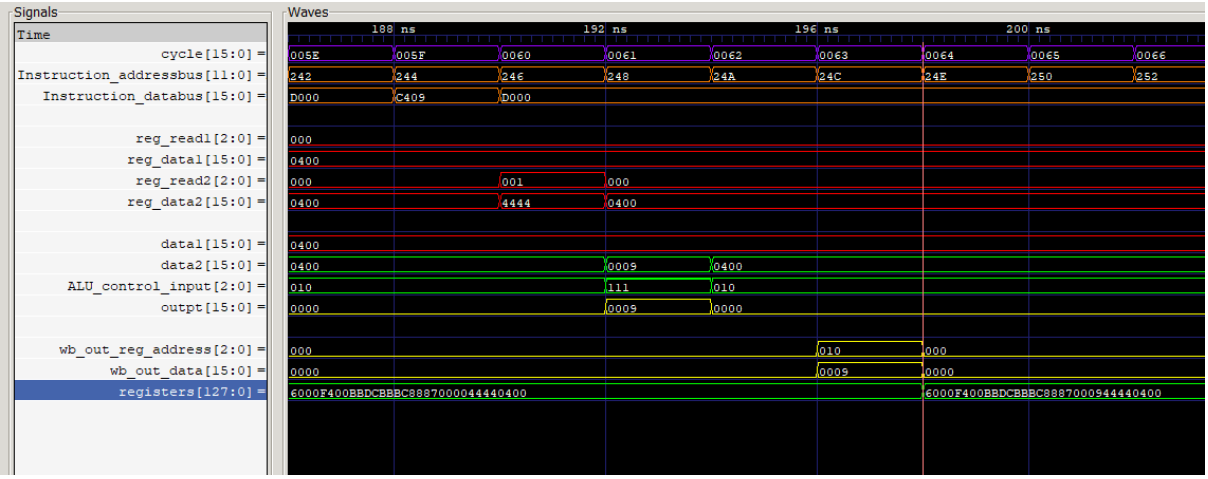
Cycle 88: This instruction does not have a memory operation.

Cycle 89: Write back to R2 = 0000h.

You can see values of registers on the Cycle 90.

MOV:

Example Instruction => C409 = 1100 010 000001001 = Mov R2, 9



Cycle 95: Instruction Fetch

Cycle 96: Instruction Decode. immediate = 0009h, Rd = R2.

Cycle 97: Execution. Outpt wire is 0009h.

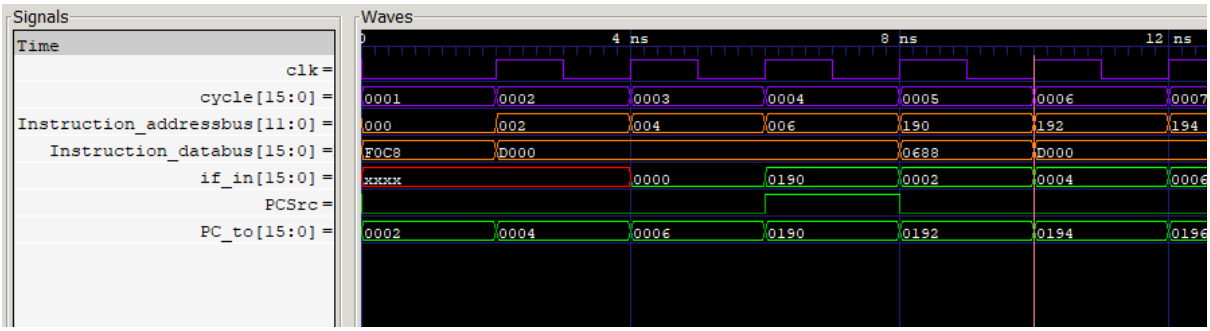
Cycle 98: This instruction does not have a memory operation.

Cycle 99: Write back to R2 = 0009h.

You can see values of registers on the Cycle 100.

JMP:

Example Instruction => F0C8 = 1111 000011001000 = Jmp 200



Cycle 1: Instruction Fetch

Cycle 2: Instruction Decode. immediate = 00C8h.

Cycle 3: Execution. Outpt wire = 200 x 2 = 0190h

Cycle 4: This instruction does not have a memory operation.

Cycle 5: This instruction does not have write back operation.

You can see the value of instruction_addressbus is changed 0190h at Cycle 6.