ASIP Laboratory - Session 2

Paul Georg Wagner Majd Mansour

May 16, 2017

Exercise 2: Using ASIP Meister

a) *What is the difference between the MicroOp implementation for `jr` and `jalr`. What is happening in hardware and when is it happening. Why has link to be global?*

The `jr` instruction jumps to the absolute address that is given in a register. The MicroOp implementation of the `jr` instruction uses the `FETCH()` macro in the IF phase and the `JUMP()` macro in the ID phase of the pipeline.

The `FETCH()` macro retrieves the next instruction and increments the program counter. It is implemented as follows.

1. Read the current program counter from the PC register.
2. Access the memory through the IMAU (instruction memory access unit) and retrieve the instruction for the current program counter.
3. Write the retrieved instruction to the instruction register (IR).
4. Increment the program counter (PC).

Since the `FETCH()` macro only has one stage, this is all done during the IF phase.

The `JUMP()` macro then jumps to the address that is stored in the specified register. It is implemented as follows.

1. Stage 1 (ID): Read the current value from the input register `rs0`, which has been specified by the instruction.
2. Stage 2 (EXE): Write the retrieved value to the program counter (PC).

The `JUMP()` macro defines two stages, so stage 1 is done in the ID phase, while stage 2 is done in the EXE phase. The write-back of the modified program counter is delayed until the EXE phase (stage 2), in order to give the `jalr` instruction a chance to read the (already incremented) program counter in the ID phase (via the `WRITELINKREG()` macro) and write it to the link register later on.

Like the `jr` instruction, the `jalr` instruction also jumps to the absolute address that is given in a register. However, it also writes the address of the next instruction (PC+1) into the link register (r31) after the `JUMP()` macro executed. The MicroOp implementation of the `jalr` instruction is very similar to the `jr` instruction. The only difference is that in the ID phase, the `WRITELINKREG()` macro is used after `JUMP()`. This macro does the following.

1. Stage 1 (ID): Read the current program counter from the PC register. The current value has already been incremented by the `FETCH()` macro in the previous IF phase).
2. Stage 4 (WB): Write the retrieved value to the link register (r31).

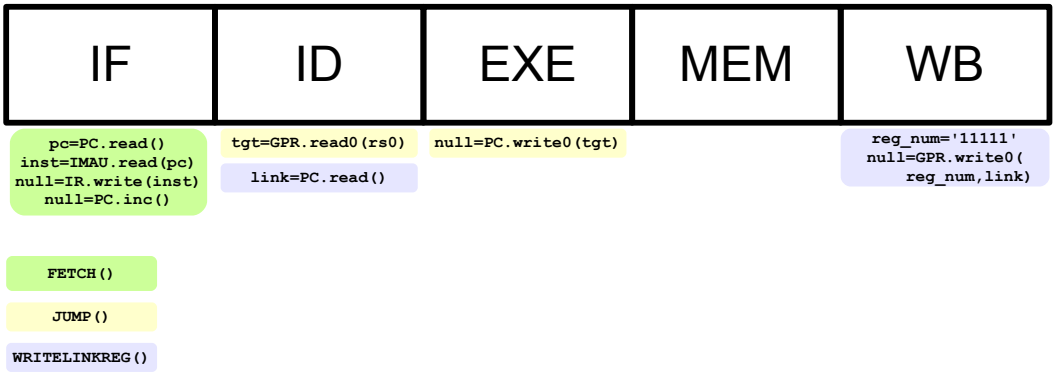


Figure 1: Pipelining of `jalr` instruction

The `JUMP()` macro defines four stages, whereas stage 1 is done in the ID phase, while stage 4 is done in the WB phase. Stages 2 and 3 do not have any content. The variable `link` has to be a global variable in the `WRITELINKREG()` macro, because it is used in different stages. More concretely, stage 1 sets the link variable and stage 4 reads it again. In order to illustrate the chronological order of the macros, figure 1 shows the sequencing of the various stages in the `jalr` instruction.

b) *What is the difference between the MicroOp implementation for* `subi` *and* `subui`*. Why this difference is only needed for the immediate instructions?*

The main difference between the implementations of `subi` and `subui` is that two different macros are used in the ID phase to retrieve the operands. The `subi` instruction uses the `GPR1READ1EXT(rs0,const)` macro, which reads the first operand from the specified register `rs0` and stores it in the variable `source0`. Then the macro uses the `EXT0` extender resource to perform a sign extension on the 16 bit immediate input value `const`. This sign extension casts the 16 bit signed immediate value to a 32 bit signed value that the ALU can operate on. The result is then stored in the variable `source1`.

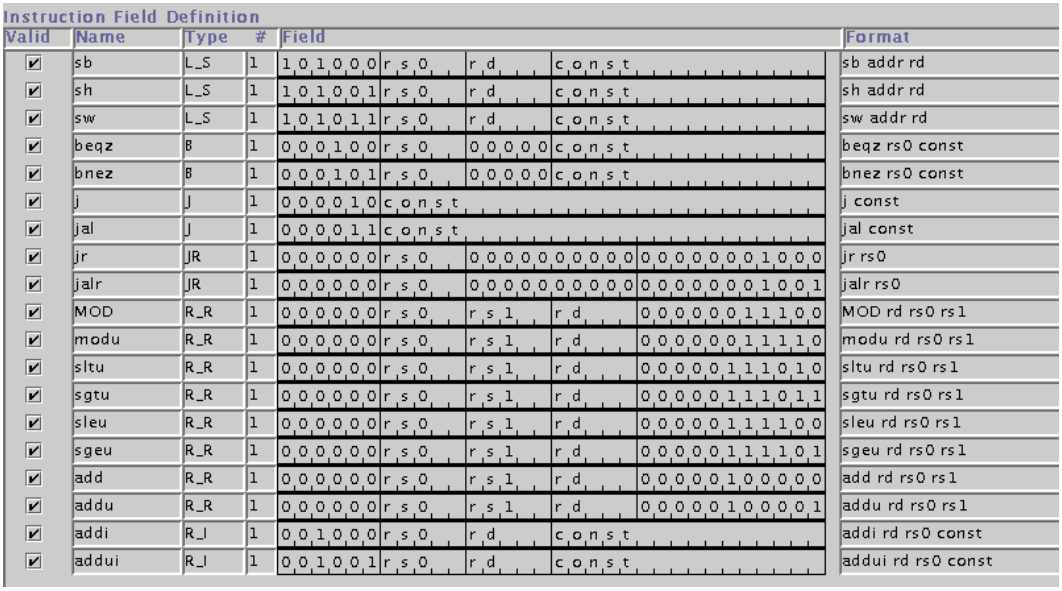
The `subui` instruction uses the macro `GPR1READ1CONST(rs0,const)` instead, which also reads the first operand from the specified register `rs0` and stores it in the variable `source0`. However, the 16 bit immediate input value `const` is now extended to 32 bits by simply setting the most significant 16 bits of the 32 bit result to zero. This is possible because we have unsigned values, so no sign extension has to be performed. The result is again stored in the variable `source1`. The `sub`/`subu` instructions do not need this distinction, because they do not load any 16 bit long immediate values. Since they directly operate on 32 bit values at all times, there is no need for a sign extension.

Exercise 3: Adding an Instruction

Adding the instructions `add`, `addi`, `addu` and `addui` takes multiple steps. First of all the required resources have to be added, so in this case an adder. However, the `ALU0` resource instance already included an `add()` as well as an `addu()` function, so this can be omitted.

In the next step, the new instructions have to be defined. For this, the `add` and `addu` instructions use the R-format, while the `addi` and `addui` immediate instructions use the I-format. The added instruction definitions in ASIPMeister are shown in figure 2.

Finally, the behavior description of the new instructions had to be added. For this, the existing macros can conveniently be utilized. The implementation of the pipeline stages for the different instructions is shown in table 1 and 2.



Instruction Field Definition

Valid	Name	Type	#	Field	Format
☑	sb	L_S	1	101000 rs0 rd const	sb addr rd
☑	sh	L_S	1	101001 rs0 rd const	sh addr rd
☑	sw	L_S	1	101011 rs0 rd const	sw addr rd
☑	beqz	B	1	000100 rs0 00000 const	beqz rs0 const
☑	bnez	B	1	000101 rs0 00000 const	bnez rs0 const
☑	j	J	1	000010 const	j const
☑	jal	J	1	000011 const	jal const
☑	jr	JR	1	000000 rs0 0000000000 0000000001000	jr rs0
☑	jalr	JR	1	000000 rs0 0000000000 0000000001001	jalr rs0
☑	MOD	R_R	1	000000 rs0 rs1 rd 0000001111100	MOD rd rs0 rs1
☑	modu	R_R	1	000000 rs0 rs1 rd 0000001111110	modu rd rs0 rs1
☑	sltu	R_R	1	000000 rs0 rs1 rd 0000011110101	sltu rd rs0 rs1
☑	sgtu	R_R	1	000000 rs0 rs1 rd 0000011110111	sgtu rd rs0 rs1
☑	sleu	R_R	1	000000 rs0 rs1 rd 0000011111100	sleu rd rs0 rs1
☑	sgeu	R_R	1	000000 rs0 rs1 rd 0000011111101	sgeu rd rs0 rs1
☑	add	R_R	1	000000 rs0 rs1 rd 0000010000000	add rd rs0 rs1
☑	addu	R_R	1	000000 rs0 rs1 rd 0000010000001	addu rd rs0 rs1
☑	addi	R_I	1	001000 rs0 rd const	addi rd rs0 const
☑	addui	R_I	1	001001 rs0 rd const	addui rd rs0 const

Figure 2: The new instructions in ASIPMeister

	add	addi
IF	FETCH()	
ID	GPR2READ(rs0,rs1)	GPR1READ1EXT(rs0,const)
EXE	ALUEXEC(add,source0,source1)	
MEM		
WB	WRITEBACK(rd,result)	

Table 1: Implementation of `add` and `addi`.

	addu	addui
IF	FETCH()	
ID	GPR2READ(rs0,rs1)	GPR1READ1CONST(rs0,const)
EXE	ALUEXEC(addu,source0,source1)	
MEM		
WB	WRITEBACK(rd,result)	

Table 2: Implementation of `addu` and `addui`.

Exercise 4: Simulating with dlxsim

a) *How many NOP's are executed as compared to the total number of instructions and as compared to the total number of executed cycles (in percentage)?*

The simulation gives a total of 150 executed operations in 183 cycles, of which 76 are NOPs. This means that $\frac{76}{150} \approx 50.7\%$ of the executed operations are NOPs, while $\frac{76}{183} \approx 41.5\%$ of the cycles are needed for NOPs.

b) *Remove the NOP after the* `add %r6, %r6, %r7` *instruction and simulate the program again by typing "make sim" and then "make dlxsim". Understand the warning, that dlxsim will print and have a look to the result array in dlxsim. What is the difference as compared to the correct result and why has it been changed in such a way?*

When executing the original version of the program, dlxsim gives the correct results as follows.

```
a+0x00 (0x00cc): 48
a+0x04 (0x00d0): 49
a+0x08 (0x00d4): 50
a+0x0c (0x00d8): 51
a+0x10 (0x00dc): 52
a+0x14 (0x00e0): 53
a+0x18 (0x00e4): 54
a+0x1c (0x00e8): 55
a+0x20 (0x00ec): 56
a+0x24 (0x00f0): 57
```

The resulting output array is the input array $[1, 2, ..., 10]$, with $C + 5 = 47$ added to each item. However, if the NOP is removed, the resulting output array changes to the following.

```
a+0x00 (0x00c8): 1
a+0x04 (0x00cc): 2
a+0x08 (0x00d0): 3
a+0x0c (0x00d4): 4
a+0x10 (0x00d8): 5
a+0x14 (0x00dc): 6
a+0x18 (0x00e0): 7
a+0x1c (0x00e4): 8
a+0x20 (0x00e8): 9
a+0x24 (0x00ec): 10
```

This is because of the data dependency between `add %r6, %r6, %r7` and `sw A(%r4), %r6`. By removing one of the three NOPs between the dependent instructions, the write-back of register `r6` happens after the operand for the store instruction has been fetched. Hence the old value of the register is stored and the addition is lost. Since the addition is responsible for adding $C + 5$ to the input array item, the new result ultimately is just the input array itself. When executing the modified program, dlxSim even warns about the unresolved dependency by printing an unresolved data dependency warning.

c) *The real CPU does not have a NOP instruction. Into which instruction is a NOP translated and what parameters (register and immediate values) does this instruction get as input? Have a look into the "TestData.IM" to find out which opcode is used to implement a NOP.*

The NOP instruction is implemented as `sll %r0, %r0, %r0`, which performs a logical left shift of length 0 on the zero register. Hence this instruction does not alter any registers. The `sll` instruction has an opcode of 6 zero bits along with a function of 11 zero bits. Furthermore the address of register `r0` is another 5 zero bits, which is why the NOP processor instruction results to 32 zero bits. This makes the NOP instruction very recognizable in machine code.

Exercise 5: Simulating with ModelSim

a) *Write a short description what is happening in the first data memory access. Mention all the data bus signals and the final register write back. In the upper part of the ModelSim waveform, you can see a clock counter. Use this value to describe when something is happening. Sometimes things happen in the middle of a clock cycle (i.e. falling edge); mention this, too.*

The first data memory access of the simulated program is the `lw %r7, C(%r0)` instruction, which loads the value of $C = 42$ into register `r7`. This instruction is fetched into the pipeline in cycle 25 (see figure 3). During this cycle, no register is written and no memory access is requested (`datareq` is 0).

In the next cycle 26, the load instruction is decoded. Simultaneously, the link register (register `r31`) is written with address $20 =$ `0x14` (see figure 4). This resembles the write-back phase of the previous instruction `jal main` at address `0x13`, which jumped to the main program in the first place. There is still no memory request in this cycle.

Finally, in cycle 28, the load instruction is in the MEM phase. Therefore, at the beginning of the cycle (rising edge of `clk`), the `datareq` signal is set to 1 (see figure 5). The memory access mode is set to *read* by setting all of the four `datawin` signals to 1 (so `datawin` is 15). Finally, the address to be read is specified using the 32 signals of `dataab`. In this case, the address is $80 =$ `0x50`.

In the midst of cycle 28 (i.e. with the falling edge of `clk`), the requested data arrives at the processor. Since the load instruction reads the value of $C = 42$ from memory, the 32 `datadb` signals resemble the value 42 (see figure 6). However, the data is not yet stable on the bus, so the `dataack` signal still shows 0.

With the rising edge of the next cycle 29, the `dataack` signal is then set to 1, indicating that the `datadb` signals are stable and can be used. This also means that the memory request has been fulfilled, which is why the `datareq` signal is reset to 0 (see figure 7). Furthermore, the load instruction has entered the WB phase of the pipeling, so the five `w_sel0` signals are set to 7, preparing the writing of register `r7`. However, the register is not yet written in this cycle, because `w_enb0` is still 0.

Finally, in the next cycle 30, the register `r7` is written with the loaded value of 42. For this, the 32 `data_in0` signals resemble the value to be written, while the `w_enb0` signal is set to 1 (see figure 8). This completes the load instruction.

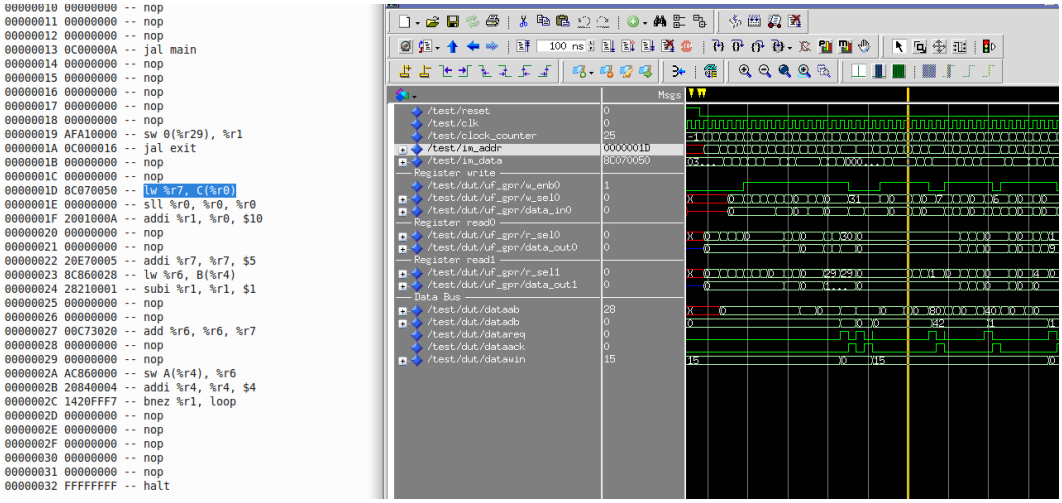


Figure 3: Cycle 25 (IF phase of load instruction).

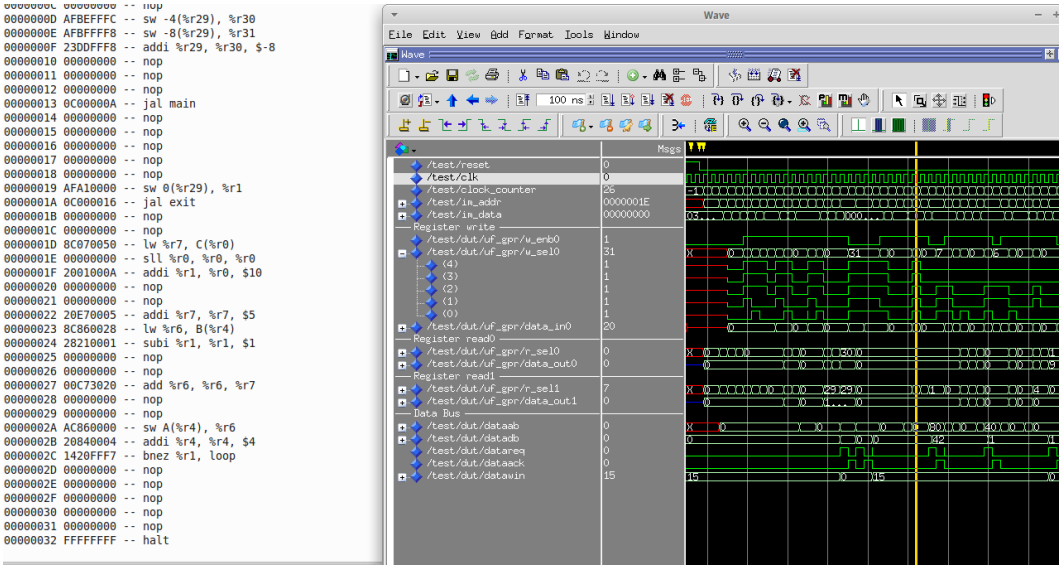


Figure 4: Cycle 26 (ID phase of load instruction).

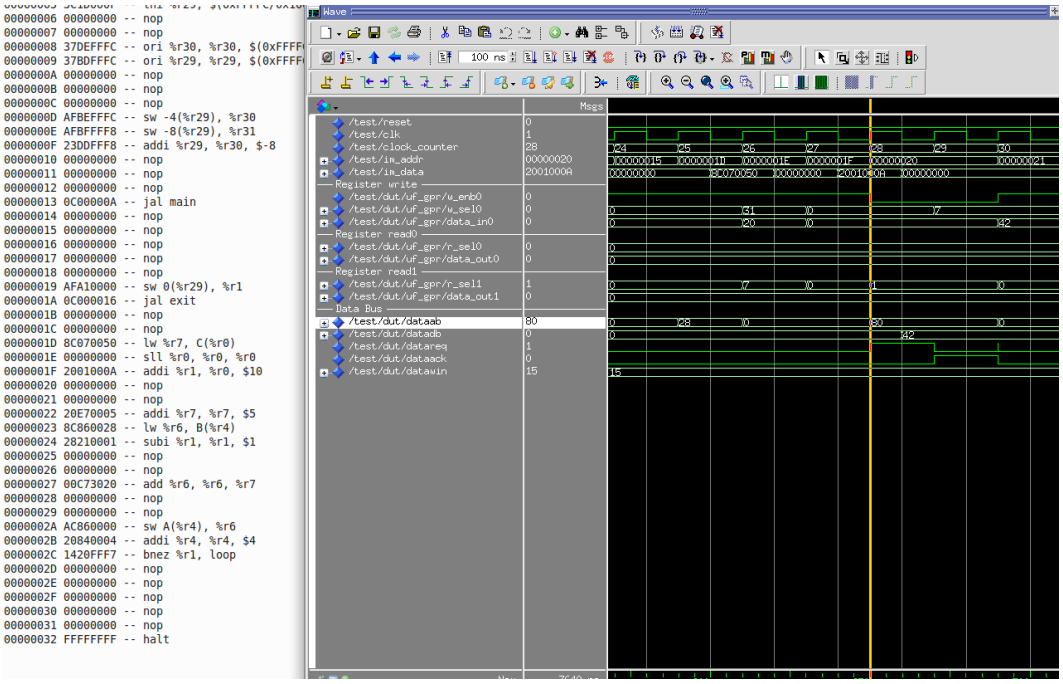


Figure 5: Begin of cycle 28 (MEM phase of load instruction).

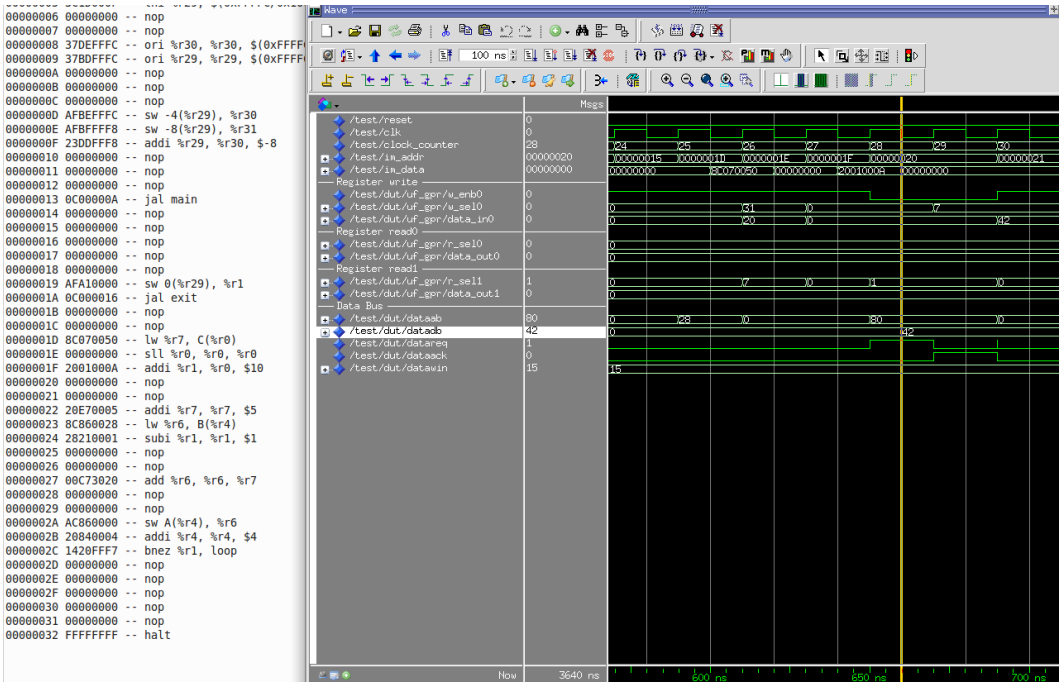


Figure 6: Mid of cycle 28 (MEM phase of load instruction).

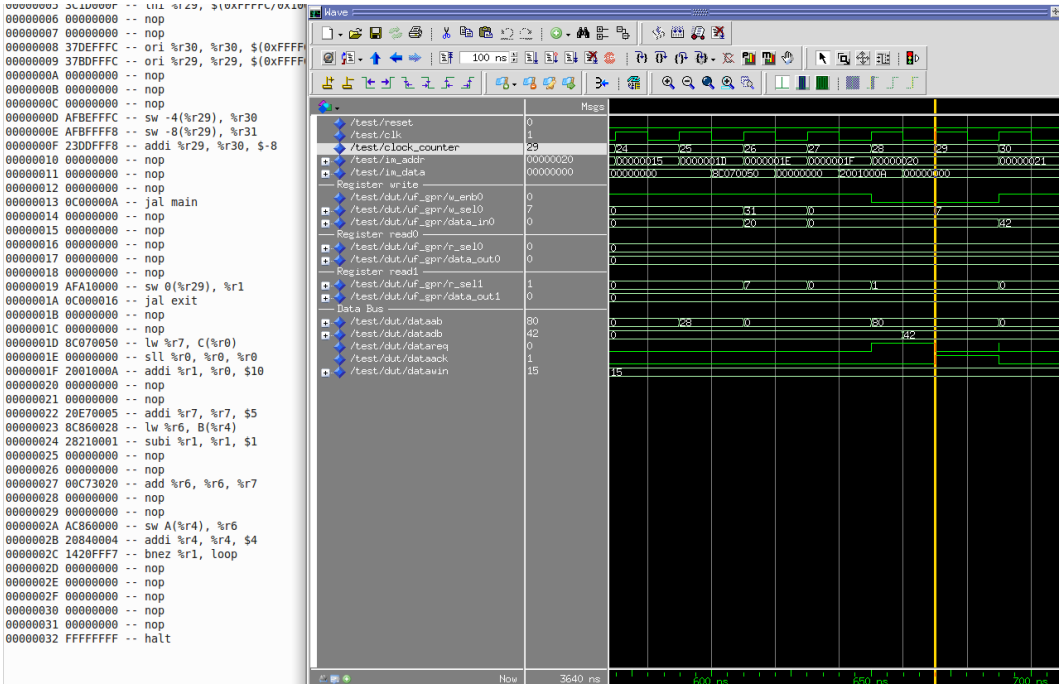


Figure 7: Cycle 29 (WB phase of load instruction).

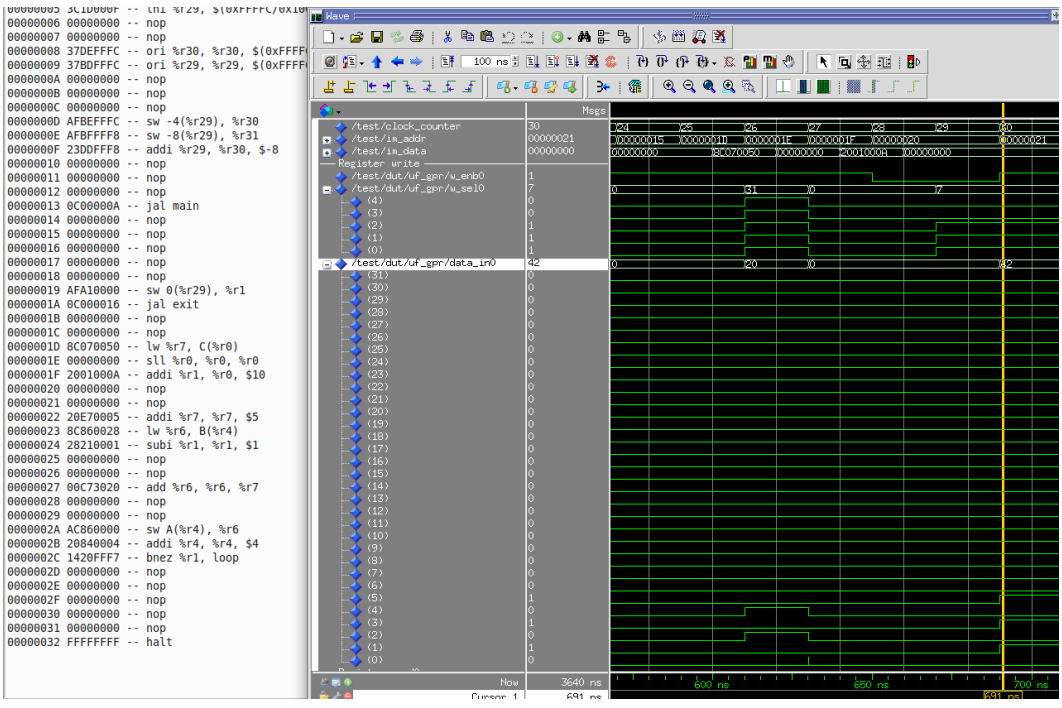


Figure 8: Cycle 30 (Load instruction has finished).