Patrick Murray

4/17/2012

EEL4713

Assignment 5

Introduction

For assignment 5 we were tasked with creating a fully functional pipelined MIPS processor. I used the code and general structure of assignment 4 to design the multi staged version in this lab. There are a number of benefits to using a pipelined datapath as opposed to a single cycle datapath, the most significant being a large speedup. We are able to produce a faster processor because we don't need to wait for every instruction to complete before beginning the next. In a single cycle processor, after we are done reading the data from a register and inputting it to the ALU we aren't doing anything else with the register until we need to write back to it. Using a pipelined version enables us to break the datapath up into five separate stages, allowing us to work on up to five instructions at once. This cuts out the slowdown of the critical path that we saw in our single cycle design.

Each one of the five stages mentioned above contains parts of the original single-cycle datapath with additional hardware needed to operate the pipelined version. The five stages consist of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB). Below is a short, general overview of each of the stages.

- IF
    - The main components in this stage are the Program Counter (PC), PC incrementer/adder, and the Instruction Memory. This stage receives signals from the MEM stage as well because that is where the branch mux is located, which supplies the PC with the next address to load.
- ID
    - This stage consists of the register file, controller, and jump mux. All the control signals are decoded and generated here. Because of this the register that is between ID and EX is very large as most of the control signals have to pass through here. This stage also contains the extending unit to either sign extend or zero extend immediate values. It receives data and signals from the WB stage because of the register file needing to be written back to.
- EX
    - This stage contains the ALU, forwarding muxes, and the JR mux. It also has a mux that takes care of the register destination which is routed through the entire pipeline and back to the ID stage to tell the register file what address to write to.
- MEM
    - This stage contains the branching logic, branch mux, data memory, and decoding unit. As mentioned previously, this is where the incremented PC signal departs from. The output of the branch mux will contain either the PC+4 signal, address we want to jump to, the address to return to after a JR, or the address to branch to.
- WB

- This stage is the smallest and only contains the mux to determine which output to route to the data of the register file in the ID stage. It essentially does exactly what its name implies, just writes all the data back to previous stages if it hasn't already been sent back.

In my datapath design I used large register files in between the five stages. Each one of these register files took in inputs from the previous stage and passed them through to the next stage. The signals that were passed through were either needed in a future stage or following the path back to the register file. I further broke these register files up into smaller files. The main register handled the unique signals going from stage to stage, while smaller register files took in the control signals only and carried them to their appropriate stages. You can see this in the sample datapath image at the end of this section.

While a pipelined design is much faster it also presents a number of new challenges. First the single cycle design needs to be converted to a pipelined datapath by adding large registers between our five cycles to stall the signals every clock cycle. This can become cumbersome because we need to pass certain signals through the registers, some signals need to avoid the registers, and some signals need to be passed backwards to over cycles from various later stages.

The main problem we need to deal with when designing the pipelined system is data hazards. There are many different hazards that need to be resolved and just figuring out which instructions create a hazard is a difficult task. A hazard can come in two general forms. The first is resolved using stalling. If we have an instruction that relies on data from another instruction which hasn't completed yet and needs to complete either completely or partially, we will need to add *nop* instructions to stall it. Stalls are mainly used when we use a load instruction (*lhu, lbu, lui, lw*) followed closely by another instruction that uses the data that is being loaded into our register file still. Stalling has the downside of slowing down our system. A *nop* is essentially just a blank instruction that does nothing, so inserting them into our datapath creates more instructions. The other hazard is resolved using forwarding. This hazard is again encountered when an instruction relies on a recent past instruction, but instead of needing to stall the entire pipeline and wasting those extra cycles, we grab the data we need from a future cycle and route it around the registers to the cycle that needs it, injecting it into the necessary signal path. This preserves our increased speedup but has the downside of being difficult to implement for the designer. I needed to again realize all the instruction combinations that would require forwarding, then build a forwarding unit into my datapath, route all the required signals into and out of it, add any additional blocks into the cycles or datapath to keep functionality, and design the logic to not only keep our initial design working but also the added forwarding and stalling.

An example of a hazard would be if the program added to values and stored the result into register 1. If the next instruction uses register 1 as a source register than the data will not yet be available to it. As I mentioned before there are a number of combinations of hazards, but primarily you need to look at the current instruction and the 3 instructions following it. If any of the 3 instructions that follow the current instruction use its value you are likely to run into a hazard. This can be slightly reduced depending on how you design your datapath. Instead of clocking my register file on the rising

edge, I instead used the falling edge. This allows you to not have to worry about any instructions that follow the current instruction by 3 spots (though you still need to take care of instruction 1 or 2 places ahead). This works by allowing the WB stage to output the data needed at the register file in time. The WB operates off of the normal rising edge clock and sends its data to the register file write port. Because the register file waits until the falling edge the data that it needs is already available and no forwarding is required.

To implement the forwarding in hardware you need a unit that handles the logic of stalls and forwards as well as a mux in front of both inputs to the ALU. Again this can vary depending on how the datapath is designed. The hazard table given to us shows that we actually need two muxes, one set in front of the ALU and the other set in front of the ID/EX register. The ID/EX muxes are only used for forwarding unsigned integers encountered during hazards with the LUI instruction. To get rid of this problem I changed the way my LUI instruction was handled. Instead of doing it inside of either the decoder or through its own logic, I passed the value straight into the ALU. If the LUI value goes into the ALU like a normal logical instruction any hazards encountered will be dealt with using the same hardware and logic used for the rest of the hazards, with the set of muxes in front of the ALU taking care of forwarding.

My initial hazard table was incomplete, as I neglected to add certain instruction combinations that would produce hazards. I had to add all of the *lui* instructions as well as the branching and jump instructions. I also made various mistakes on the correct forwarding logic for certain hazards. Below is the hazard table I used to design my forwarding and stalling units. This was provided to the class.

	R	AI	L	S	JR	LUI	B	JAL
R	1a: exmem.rd=idex.rs(1) =idex.rt(2) 2a: memwb.rd=idex.rs(3) =idex.rt(4) 3a: memwb.rd=ifid.rs(5) =ifid.rt(6)	1a: exmem.rd=idex.rs(1) 2a: memwb.rd=idex.rs(3) 3a: memwb.rd=ifid.rs(5)	1a: exmem.rd=idex.rs(1) 2a: memwb.rd=idex.rs(3) 3a: memwb.rd=ifid.rs(5)	1a: exmem.rd=idex.rs(1) =idex.rt(2) 2a: memwb.rd=idex.rs(3) =idex.rt(4) 3a: memwb.rd=ifid.rs(5) =ifid.rt(6)	1a: idex.rd=ifid.rs(stall) 2a: exmem.rd=ifid.rs(8) 3a: memwb.rd=ifid.rs(5)		1a: idex.rd=ifid.rs(stall) =ifid.rt(stall) 2a: exmem.rd=ifid.rs(8) =ifid.rt(10) 3a: memwb.rd=ifid.rs(5) =ifid.rt(6)	
AI	1a: exmem.rt=idex.rs(1) =idex.rt(2) 2a: memwb.rt=idex.rs(3) =idex.rt(4) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	1a: exmem.rt=idex.rs(1) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=ifid.rs(5)	1a: exmem.rt=idex.rs(1) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=ifid.rs(5)	1a: exmem.rt=idex.rs(1) =idex.rt(2) 2a: memwb.rt=idex.rs(3) =idex.rt(4) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	1a: idex.rt=ifid.rs(stall) 2a: exmem.rt=ifid.rs(8) 3a: memwb.rt=ifid.rs(5)		1a: idex.rt=ifid.rs(stall) =ifid.rt(stall) 2a: exmem.rt=ifid.rs(8) =ifid.rt(10) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	
L	1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: memwb.rt=idex.rs(3) =idex.rt(4) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	1a: ifid.rt=pre.rs(stall) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=ifid.rs(5)	1a: ifid.rt=pre.rs(stall) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=ifid.rs(5)	1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: memwb.rt=idex.rs(3) =idex.rt(4) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	1a: ifid.rt=pre.rs(stall) 2a: idex.rt=pre.rs(stall) 3a: memwb.rt=ifid.rs(5)		1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: idex.rt=pre.rs(stall) =pre.rt(stall) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	
S								
JR								
LUI	1a: exmem.rt=idex.rs(11) =idex.rt(12) 2a: memwb.rt=idex.rs(13) =idex.rt(14) 3a: memwb.rt=ifid.rs(15) =ifid.rt(16)	1a: exmem.rt=idex.rs(11) 2a: memwb.rt=idex.rs(13) 3a: memwb.rt=ifid.rs(15)	1a: exmem.rt=idex.rs(11) 2a: memwb.rt=idex.rs(13) 3a: memwb.rt=ifid.rs(15)	1a: exmem.rt=idex.rs(11) =idex.rt(12) 2a: memwb.rt=idex.rs(13) =idex.rt(14) 3a: memwb.rt=ifid.rs(15) =ifid.rt(16)	1a: idex.rt=ifid.rs(17) 2a: exmem.rt=ifid.rs(18) 3a: memwb.rt=ifid.rs(15)		1a: idex.rt=ifid.rs(17) =ifid.rt(19) 2a: exmem.rt=ifid.rs(18) =ifid.rt(20) 3a: memwb.rt=ifid.rs(15) =ifid.rt(16)	
B								
JAL	1a: exmem.31=idex.rs(21) =idex.rt(22) 2a: memwb.31=idex.rs(23) =idex.rt(24) 3a: memwb.31=ifid.rs(25) =ifid.rt(26)	1a: exmem.31=idex.rs(21) 2a: memwb.31=idex.rs(23) 3a: memwb.31=ifid.rs(25)	1a: exmem.31=idex.rs(21) 2a: memwb.31=idex.rs(23) 3a: memwb.31=ifid.rs(25)	1a: exmem.31=idex.rs(21) =idex.rt(22) 2a: memwb.31=idex.rs(23) =idex.rt(24) 3a: memwb.31=ifid.rs(25) =ifid.rt(26)	1a: idex.31=ifid.rs(27) 2a: exmem.31=ifid.rs(28) 3a: memwb.31=ifid.rs(25)		1a: idex.31=ifid.rs(27) =ifid.rt(29) 2a: exmem.31=ifid.rs(28) =ifid.rt(30) 3a: memwb.31=ifid.rs(25) =ifid.rt(26)	

Figure 1 - Hazard table

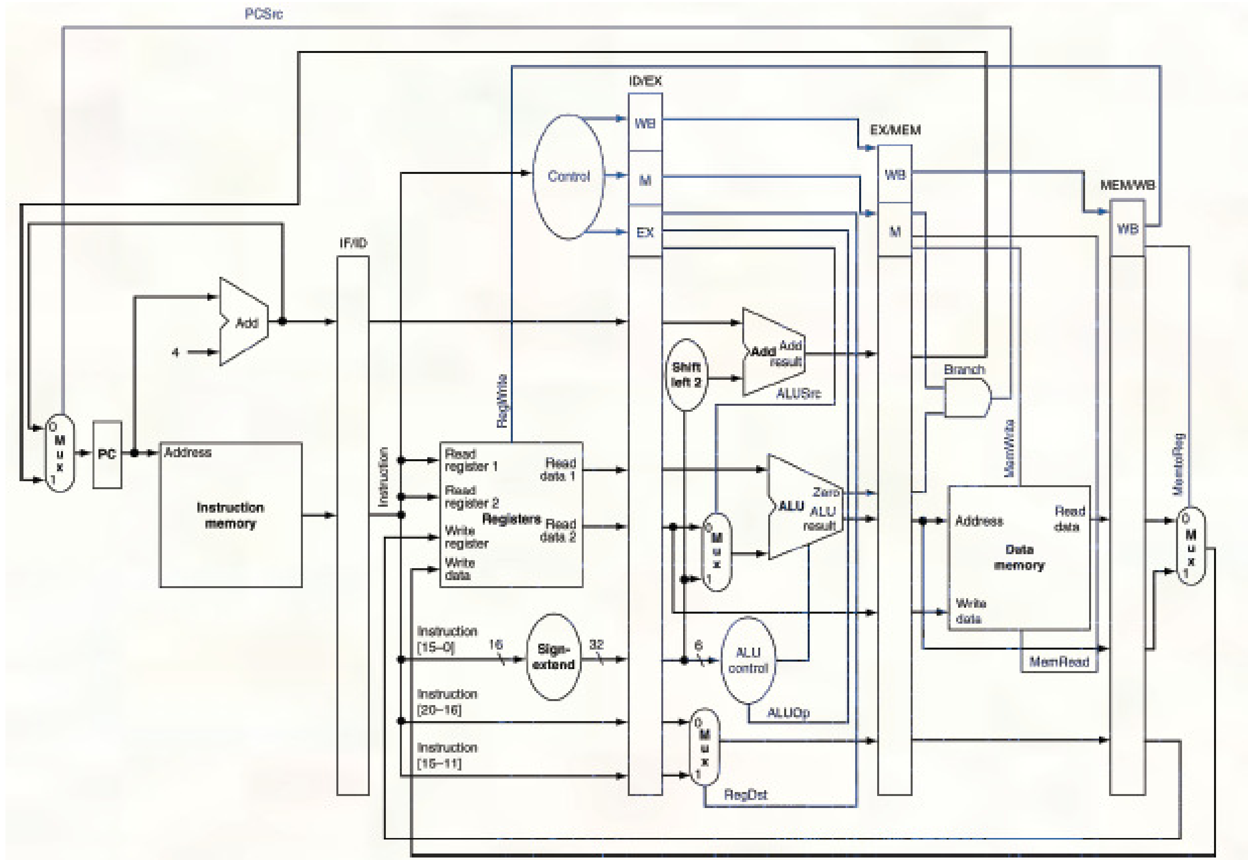


Figure 2 - Simplified view of the pipelined datapath and stages

Book Questions

Using the blue book.

4.12.1

a) Pipelined = 500ps, non-pipelined = (500+100+350+400+300) = 1650ps
b) Pipelined = 200ps, non-pipelined = (200+150+120+190+140) = 800ps

4.12.2

a) Pipelined = 1650ps (with forwarding), non-pipelined = 1650ps
b) Pipelined = 800ps (with forwarding), non-pipelined = 800ps

4.12.3

Split the highest latency stage
a) MEM, new time = 400ps
b) IF, new time = 190ps

4.20.1

<table>
<tr><td>a</td><td>lw followed by add (2 apart)
lw followed by sw (3 apart)
add followed by sw (1 apart)
add followed by add (1 apart)
add followed by sw (2 apart)</td></tr>
<tr><td>b</td><td>add followed by sw (1 apart)
add followed by lw (2 apart)
sw followed by lw (1 apart)
sw followed by add (2 apart)</td></tr>
</table>

4.20.2

<table>
<tr><td>a</td><td>With forwarding
none
Without forwarding
lw followed by add (2)
add followed by add (1)
add followed by sw (2)
add followed by sw (1)</td></tr>
<tr><td>b</td><td>With
lw followed by add (1)
Without forwarding
add followed by sw (1)
lw followed by add (1)</td></tr>
</table>

4.24.1

a) Always taken = 75%, never taken = 25%
b) Always taken = 60%, never taken = 40%

4.24.2

a) 0%
b) 25%

2.24.3

a) 75%
b) 60%

Section 3: Instruction

JR

Taken from lab4demo.mif file.

Instruction:
lui 7,0x0000
ori 7,7,0x308
jr 7

This will store 0x00000308 into register 7 and then jump to this address. Forwarding is needed because register 7 does not have the entire 0x00000308 value when JR needs it, so lui is forwarded for the ori instruction, and then that is forwarded for the JR instruction.

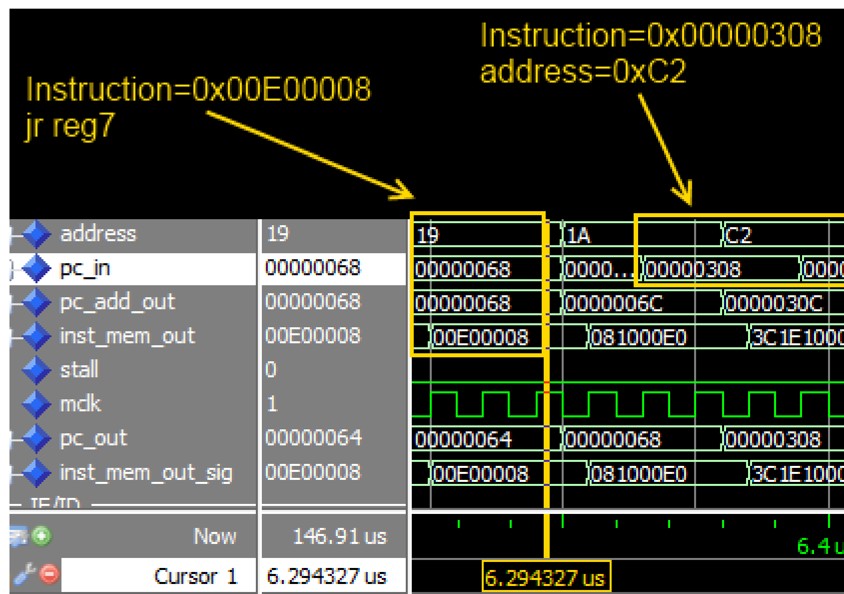


Figure 3 - JR going to contents of register 7

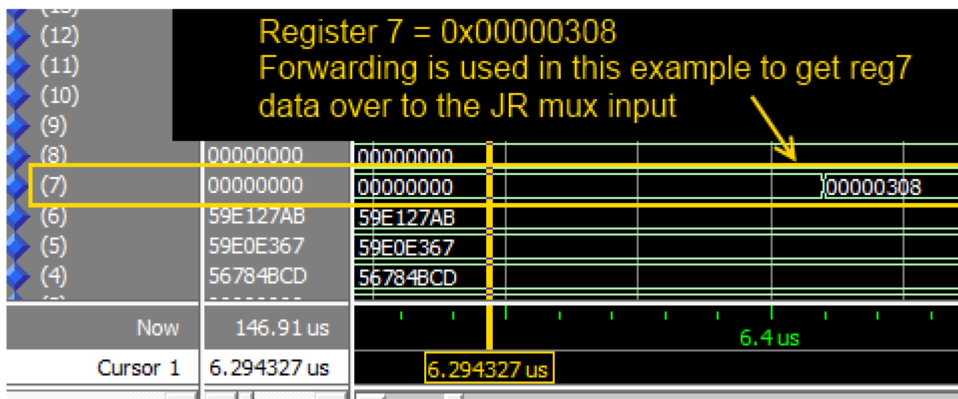


Figure 4 - JR, showing contents of register 7

Logical Functions

Test Code MIF
`lui   $a0,  $zero, 0x2568`
`lui   $a1, $zero, 0x3a97`
`ori   $a0,  $a0, 0xabef`
`ori   $a1, $a1, 0x5be0`
`and   $a2,$a0,$a1`
`andi  $a3,$a0,0x4510`
`or    $s0,$a0,$a1`
`nor   $s1,$a0,$a1`
`sll   $s2,$a0,0x4`
`srl   $s3,$a0,0x2`

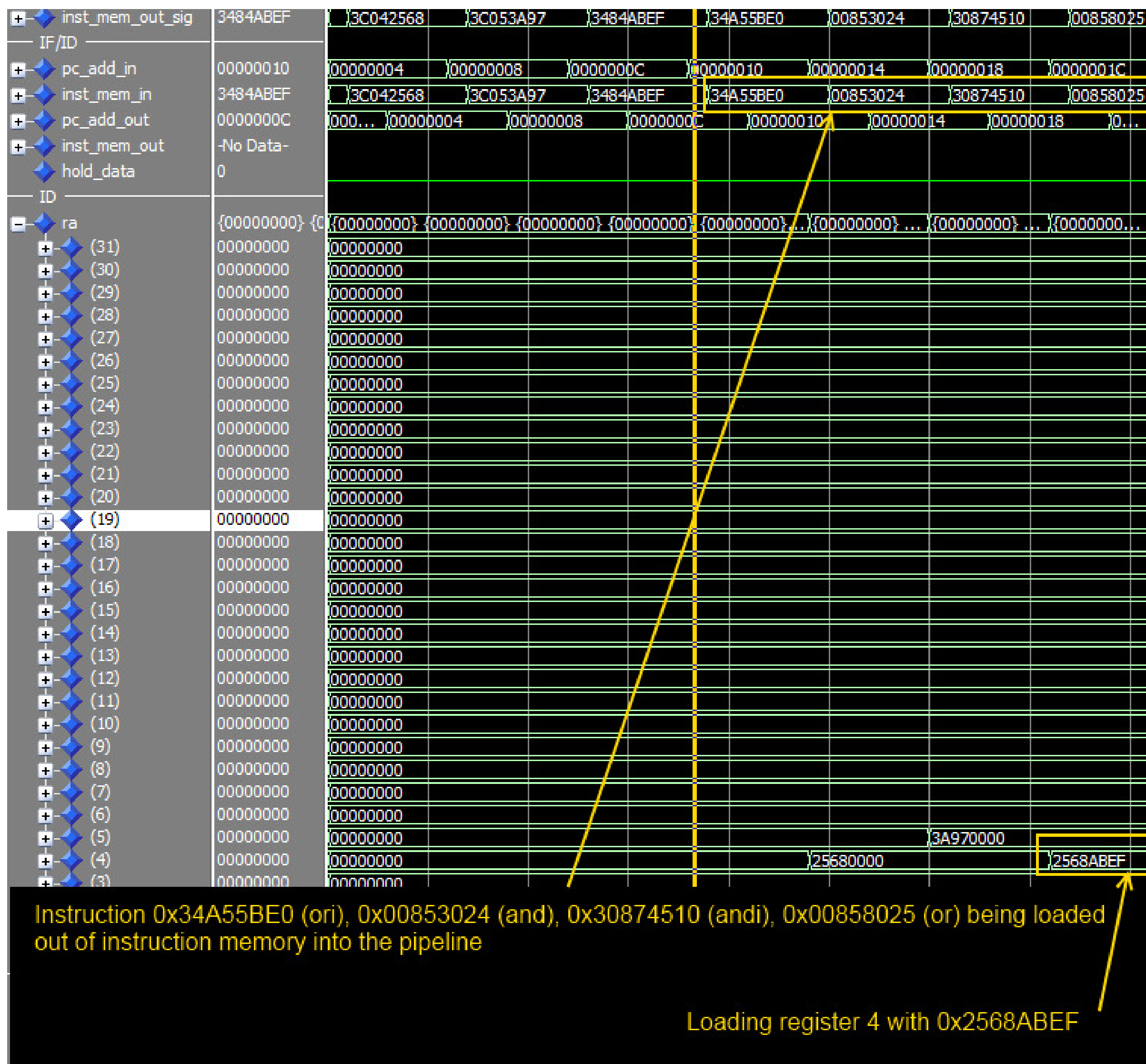


Figure 5 - Logical function test

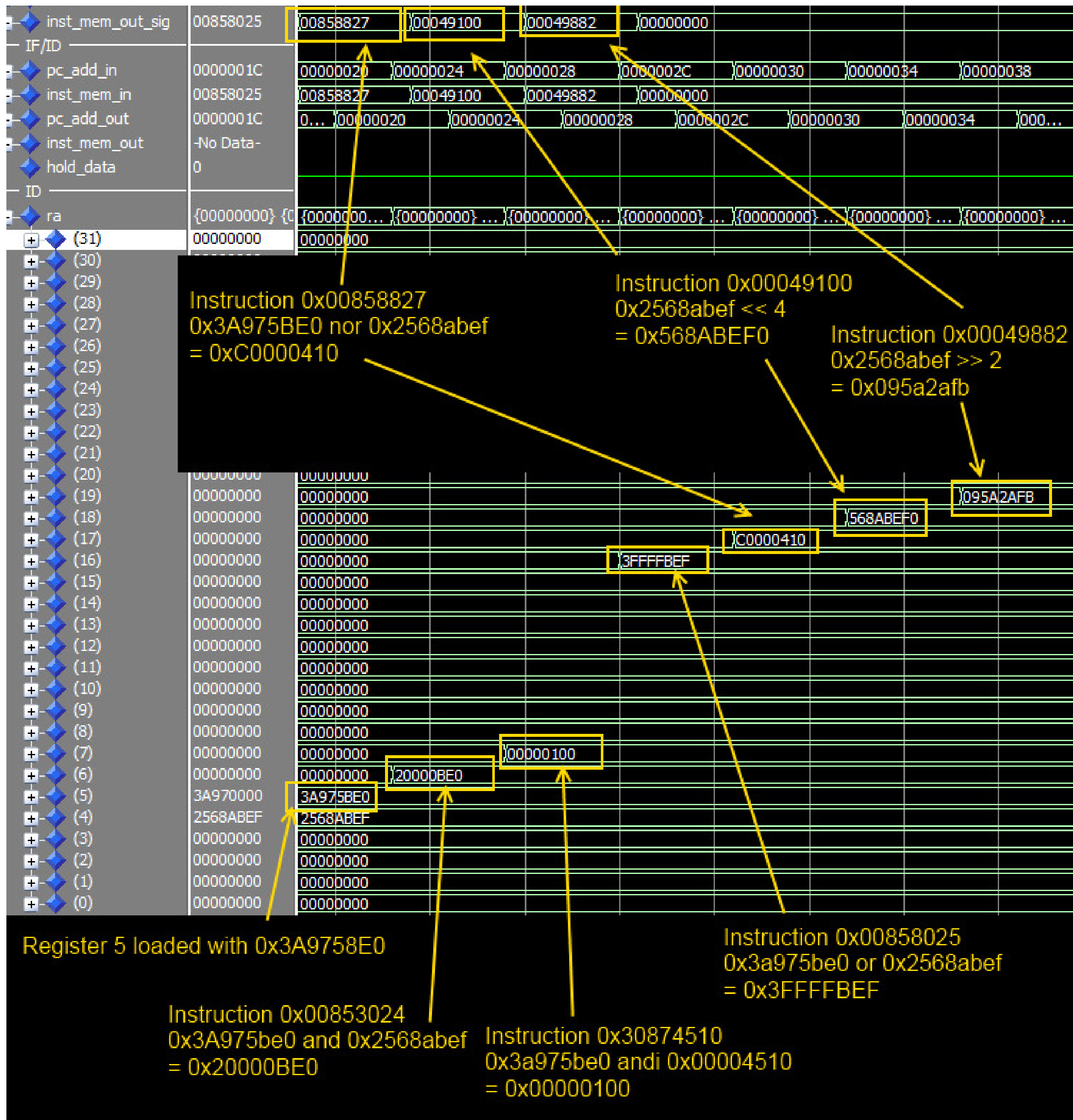


Figure 6 - Logical function test

BEQ/BNE

Test Branch MIF	
ori $s0,$0,0x1000	branch1:
ori $s1,$0,0x2000	bne $s0,$s2,end
ori $s2,$0,0x1000	bne $s0,$a1,end
beq $s0,$s1,branch1	lui $s5,$zero,0xFFFF
beq $s0,$s2,branch1	end:
lui $s4,$zero,0xFFFF	lui $s6,$zero,0xFFFF

Instruction 0x12110003
Branch if reg16 = reg17
This is not true here
so continue on

Instruction 0x12120002
Branch if reg16 = reg18
This is true so branch ahead
skipping the next address.
This is verified because the
next instruction isn't executed
(lui $s4, 0xFFFF)

Instruction 0x16120003
Branch if reg16 /= reg18
This is false so continue on

Instruction 0x16050002
Branch if reg16 /= reg17,
which is true, so again
branch ahead skipping
the next instruction
(lui $s5, 0xFFFF)

Figure 7 - Branch testing

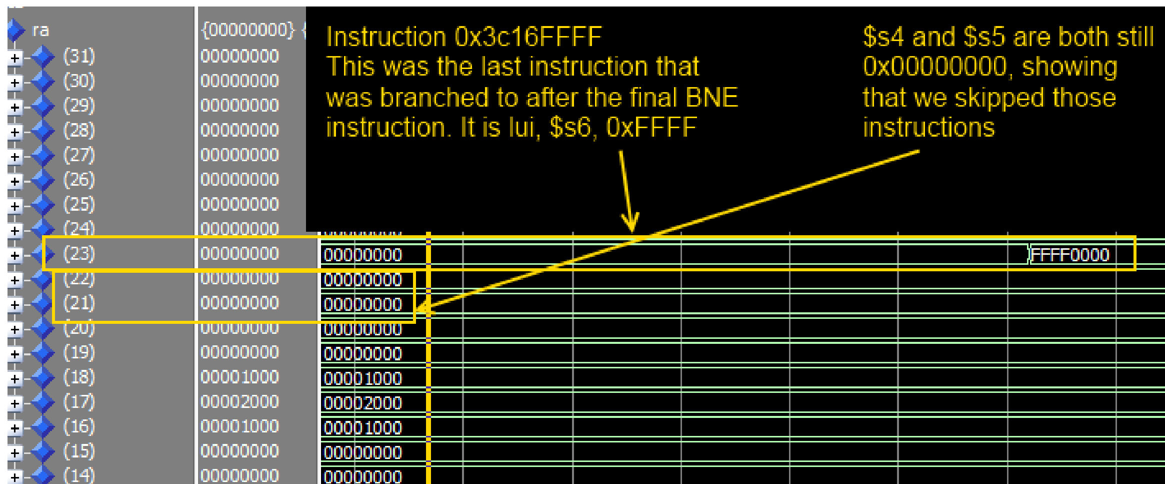


Figure 8 - Branch testing outcome

SLT, SLTU, SLTI, SLTIU

SLT/SLTU/SLTI/SLTIU Test MIF
lui $a0,0xFF11
lui $s0,0xFFFF
ori $s0,$s0,0xFFFF
ori $s1,$s1,0x1234
ori $s2,$s2,0x2345
slt $t0,$s1,$s2
slt $t1,$s2,$s1
sltu $t3,$s0,$s1
sltu $t4,$s1,$a0
slti $t5,$s1,0x4000
slti $t6,$s2,0x1900
sltiu $t7,$a0,-6
sltiu $t8,$a0,0x4161

The difference between SLT and SLTU as well as SLTI and SLTIU is how the code perceives the data. Because my main example images didn't show the differences well enough I created another short MIF file to test the signed and unsigned version against two identical numbers. This can be seen and explained in the image below. The same concept applies to SLTI vs SLTIU as well.

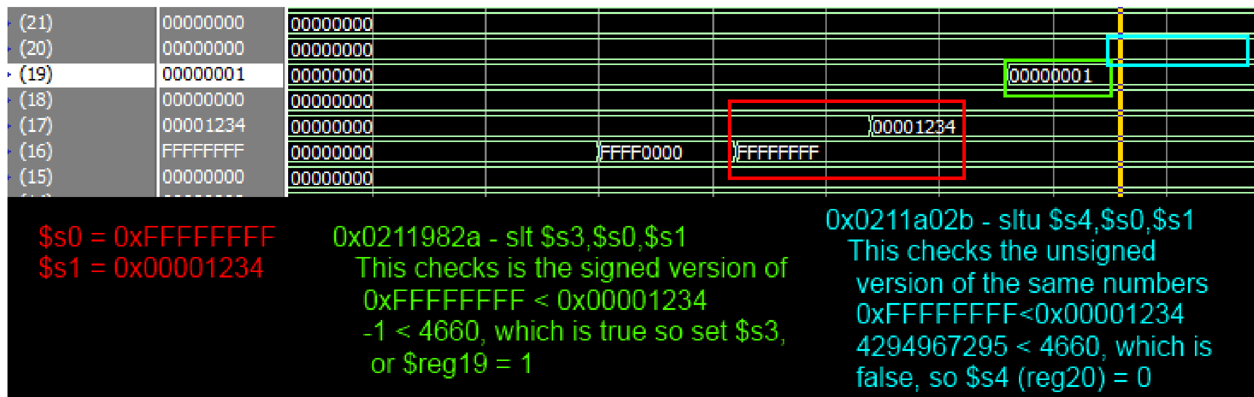


Figure 9 - Signed vs Unsigned

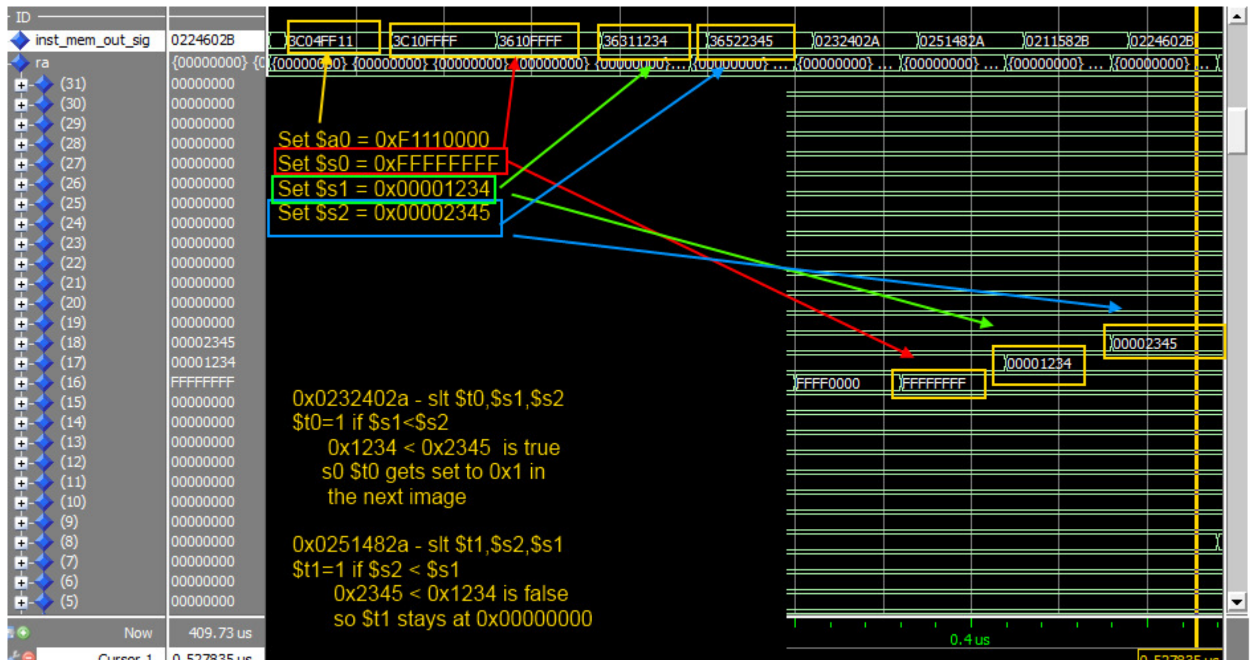


Figure 10 - SLT testing

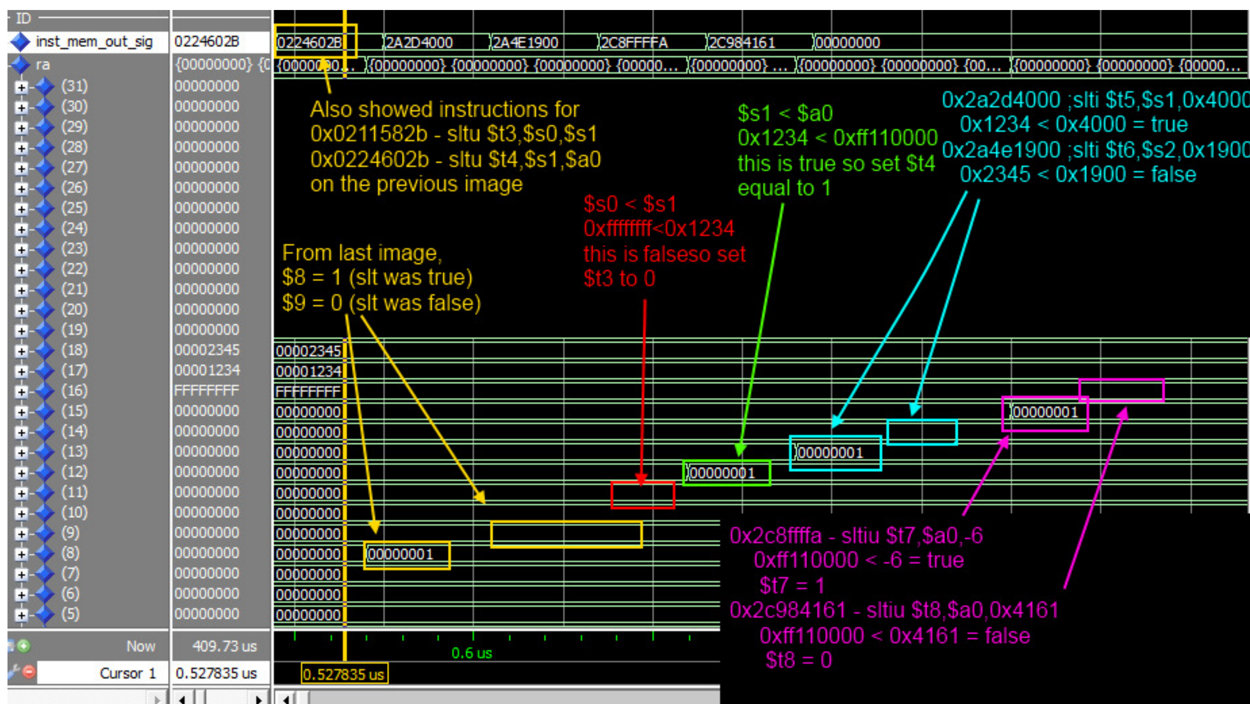


Figure 11 - SLT testing

SUB, SUBU, ADD, ADDU

The main difference between the ADD vs ADDU and SUB vs SUBU is that both of signed versions produce overflow while the unsigned version do not.

Test MIF code
lui $s0,0x8193
addi $s1, $s1,0x2468
addi $s2,$s2,0x1234
add $t0,$s1,$s2
addu $t1,$s0,$s1
addu $t2,$s1,$s2
sub $t3,$s1,$s0
sub $t4,$s1,$s2
subu $t5,$s1,$s0
subu $t6,$s1,$s2

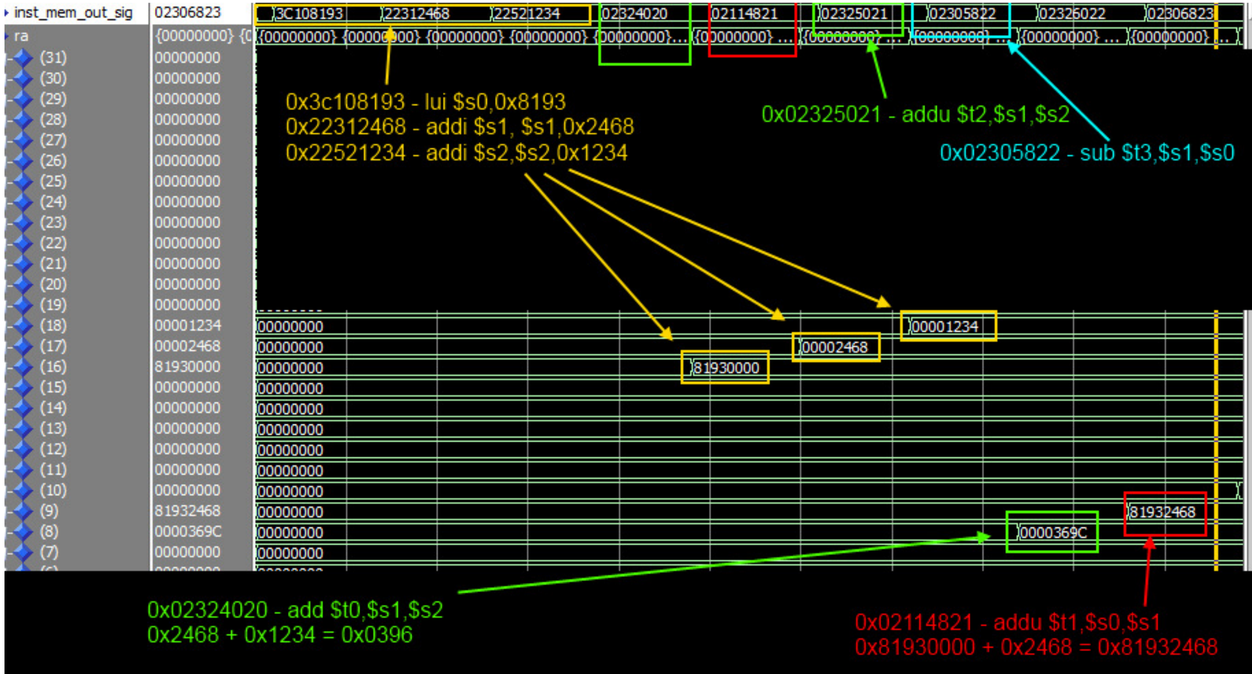


Figure 12 - SUB/SUBU/ADD/ADDU

Figure 13 - SUB/SUBU/ADD/ADDU

SW, SH, SB

Test Code MIF
lui $9,0x1000
lui $15,0x0000
ori $15,$15,0x0000
sw $15,0($9)
sw $15,4($9)
sw $15,8($9)
lui $10,0xABCD
ori $10,$10,0x1234
lui $2,0xffff
lui $3,0xffff
sb $10,0($9)
sh $10,4($9)
sw $10,8($9)
lw $11,0($9)
lw $12,4($9)
lw $13,8($9)

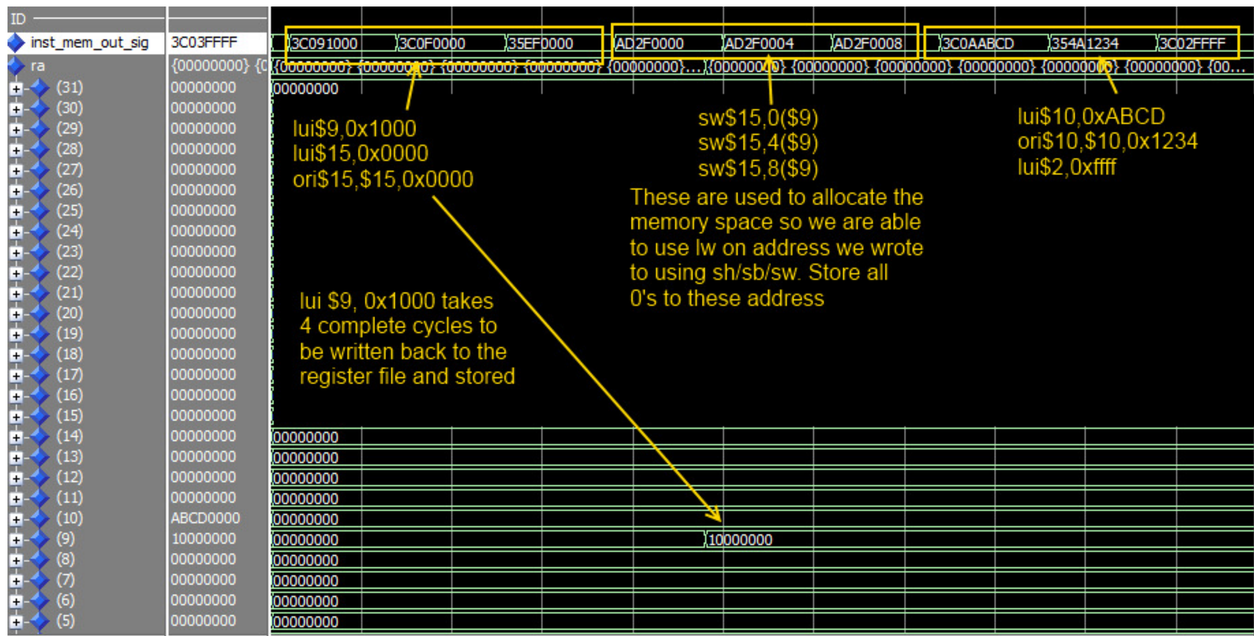


Figure 14 - Setting up the registers for the store test

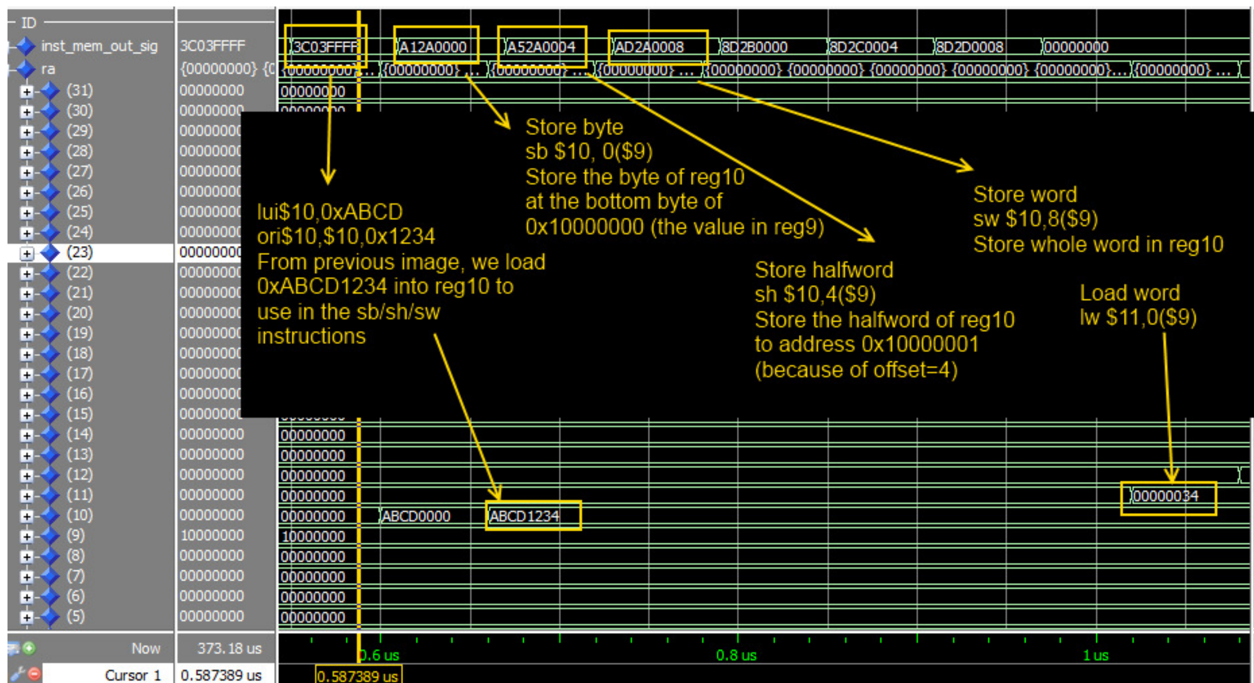


Figure 15 - Storing the values using SB, SH, and SW

Figure 16 - Using LW to read the values from memory after using SB, SH, and SW

LW, LHU, LBU

Test Code MIF
`lui    $s0,0x1000`
`lui    $t0,$zero,0xbad2`
`ori    $t0,$t0,0xbeef`
`ori    $s2,$s2,0x0000`
`ori    $s2,$s2,0x0000`
`sw     $t0,0($s0)`
`lw     $t1,0($s0)`
`lhu    $t2,0($s0)`
`lbu    $t3,0($s0)`

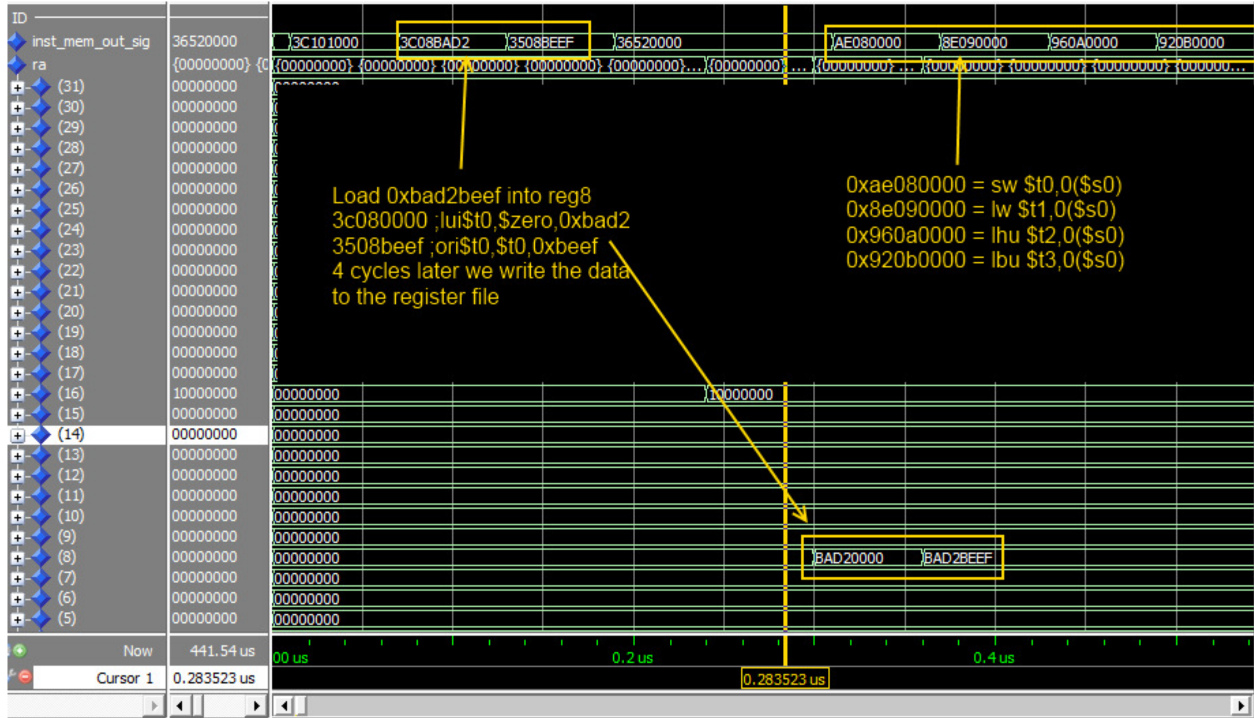


Figure 17 - Setting up registers to test LW, LHU, and LBU. Also shows the instructions in the IF stage

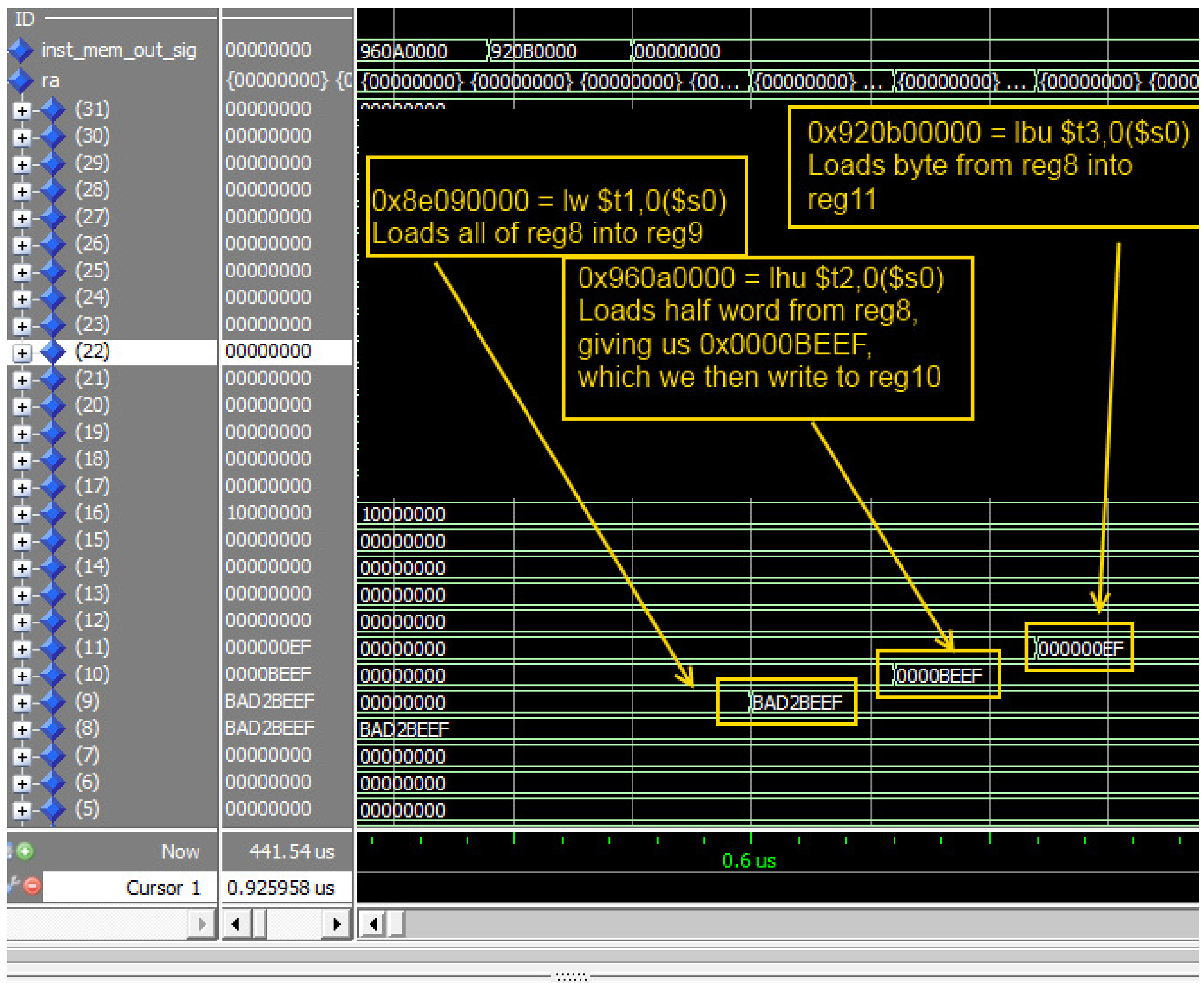
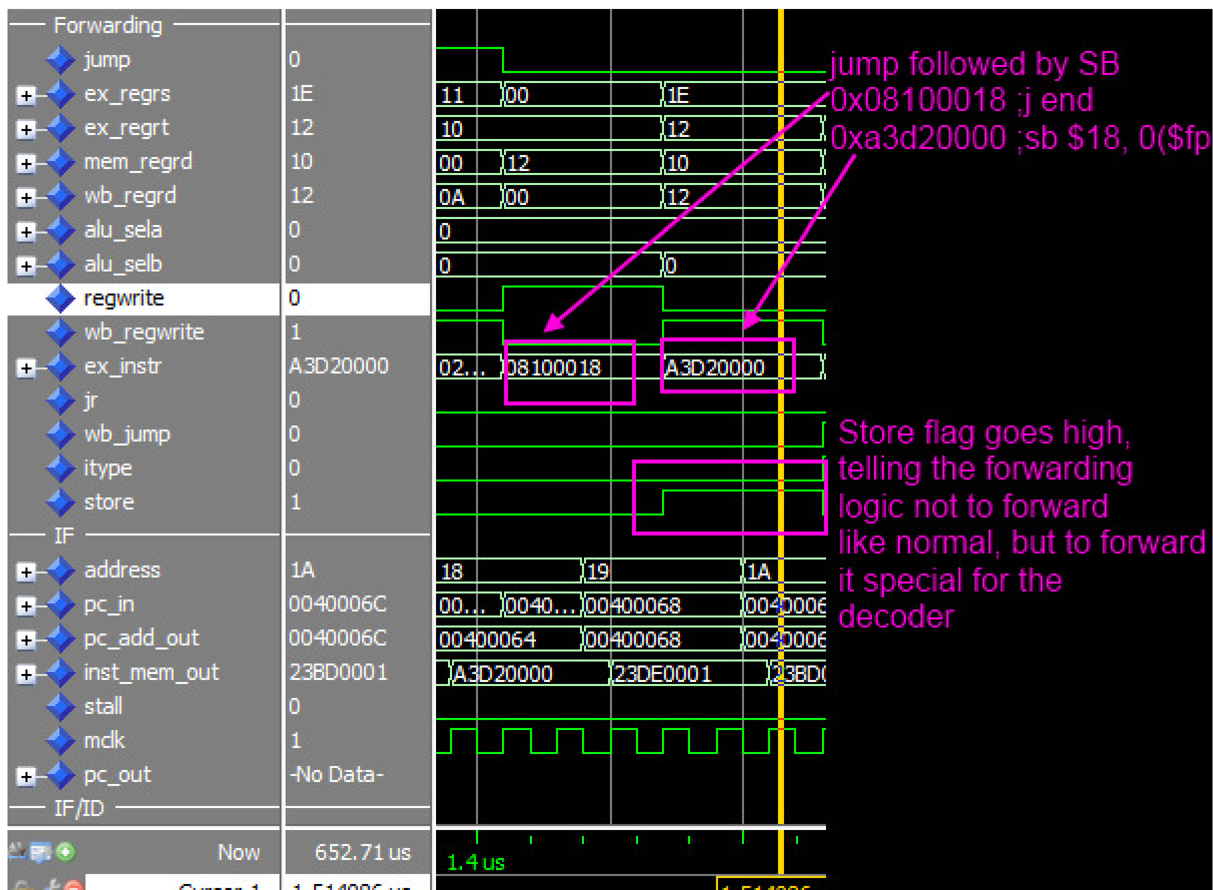


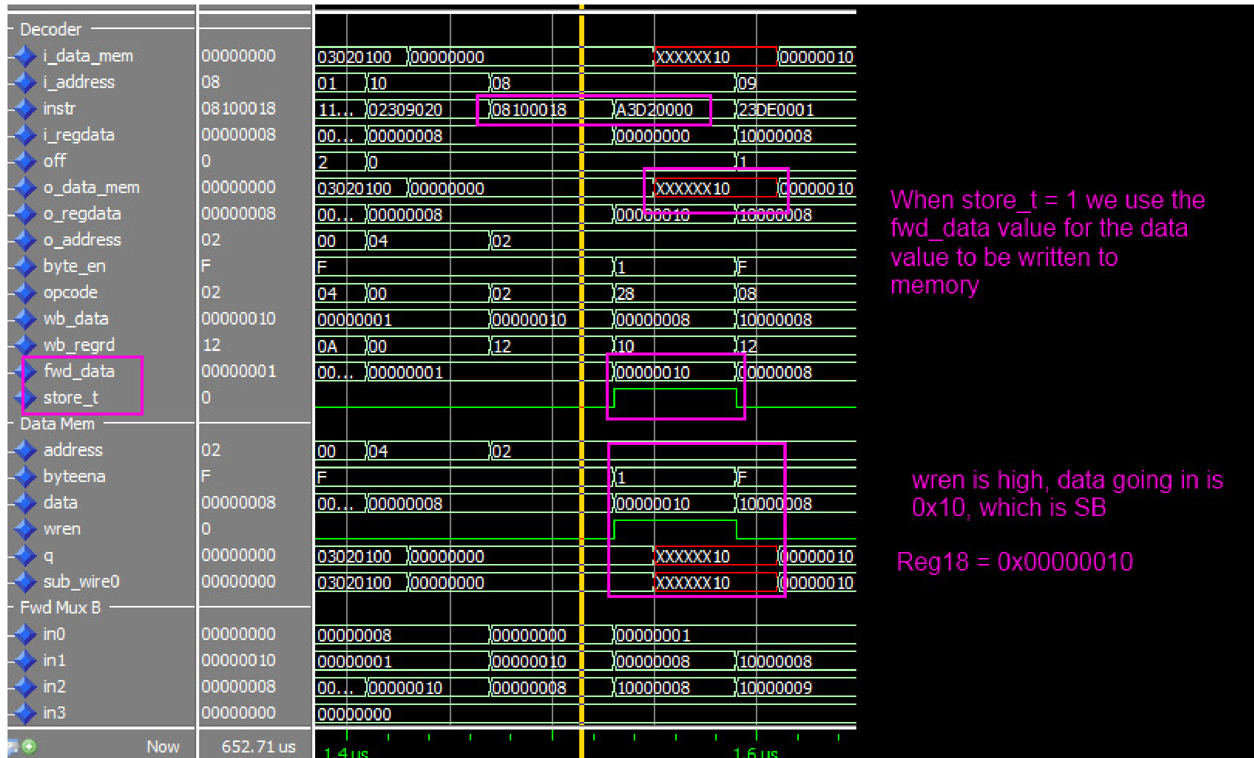
Figure 18 - Register values from the result of LW, LHU, and LBU

Section 4: Hazard detection and forwarding

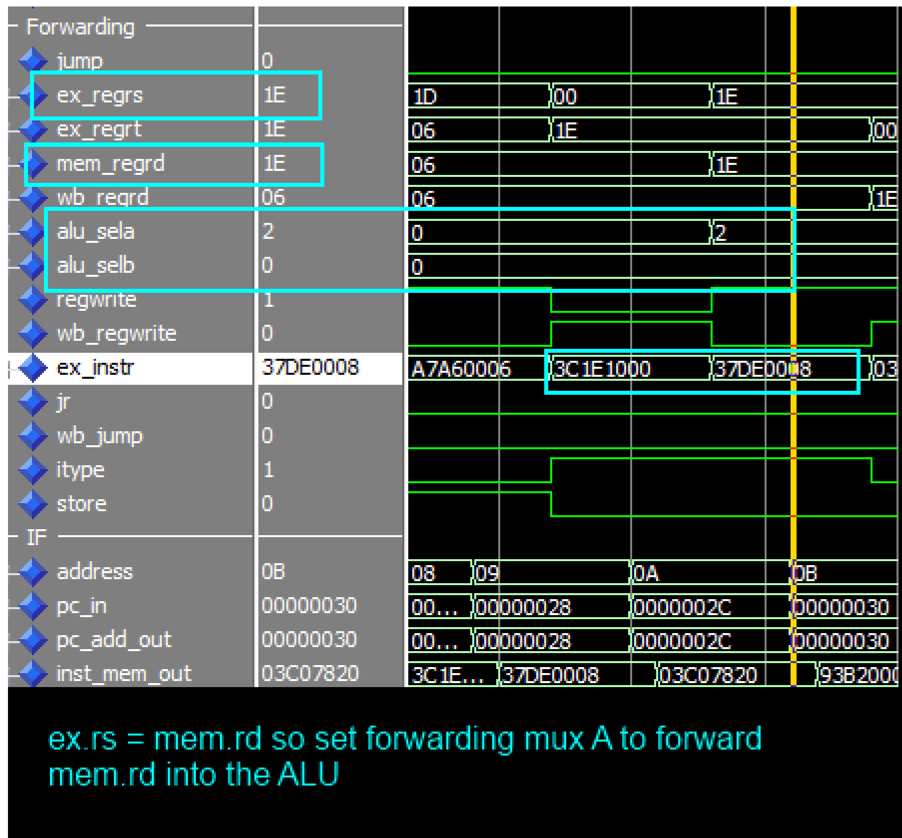
I got this working 2 hours before the extended deadline, sorry for the quick images and short descriptions, I didn't plan to be working on it this long.

I had issues forwarding with my decoder so even though the forwarding worked fine for normal instructions, it wasn't working 100% with load and store instructions. The way I did forwarding for those was inside the actual decoder and partially inside the forward unit as well. In the forward unit I watched to see if the last instruction had a hazard (so if ex.rt/ex.rs = mem.rd). If so I sent the data from that register that needed to be forwarded along with a 1 bit flag signal through the pipeline, into MEM stage, and into the decoder. If the flag went true I took that forwarded data as the input, otherwise I took the normal data from the pipeline. I also routed the wb.rd as well as the wb.data into the decoder. I watched this to see if mem.rd = wb.rd, and if it did I forwarded, or used the wb.data I had brought over. This was not a very fun or clean way to do this, but the traditional ways were giving me a lot of problems. Everytime I would fix one issue another would come up and I was already behind from other problems with the pipelining and previous checkpoints.

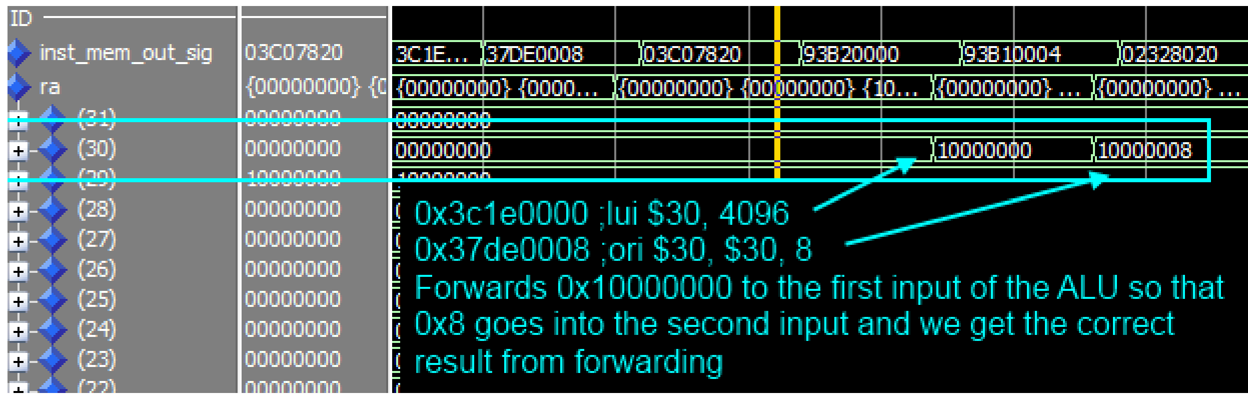



Decoder
i_data_mem
i_address
instr
i_regdata
off
o_data_mem
o_regdata
o_address
byte_en
opcode
wb_data
wb_regrd
fwd_data
store_t
Data Mem
address
byteena
data
wren
q
sub_wire0
Fwd Mux B
in0
in1
in2
in3
Now
652.71 us
1.4 us
1.6 us
When store_t = 1 we use the fwd_data value for the data value to be written to memory
wren is high, data going in is 0x10, which is SB
Reg18 = 0x00000010

Reg18 contains 0x00000010, this is used with SB to the first byte, so we write 0x10 to the bottom byte of that address



Forwarding
jump 0
ex_regrs 1E
ex_regrt 1E
mem_regrd 1E
wb_regrd 06
alu_sela 2
alu_selb 0
regwrite 1
wb_regwrite 0
ex_instr 37DE0008
jr 0
wb_jump 0
itype 1
store 0
IF
address 0B
pc_in 00000030
pc_add_out 00000030
inst_mem_out 03C07820
ex.rs = mem.rd so set forwarding mux A to forward mem.rd into the ALU



ID
inst_mem_out_sig 03C07820
ra
(30) 00000000
10000000
10000008
0x3c1e0000 ;lui $30, 4096
0x37de0008 ;ori $30, $30, 8
Forwards 0x10000000 to the first input of the ALU so that 0x8 goes into the second input and we get the correct result from forwarding

Forward mux A, which outputs into the top ALU input, is told to forward mem.rd by sel=2
So it outputs 0x10000000

lbu $18, 0($29) = 0x0000
lbu $17, 4($29) = 0x0008
add $16, $17, $18
=0x0000+0x0008 = 0x0008

slti $10, $16, 11
0x08 < 11 is true, so
$10 = 0x00000001

$17=0x0008
$16=0x0008
$18=0x08+0x08=0x10

add $18, $17, $16

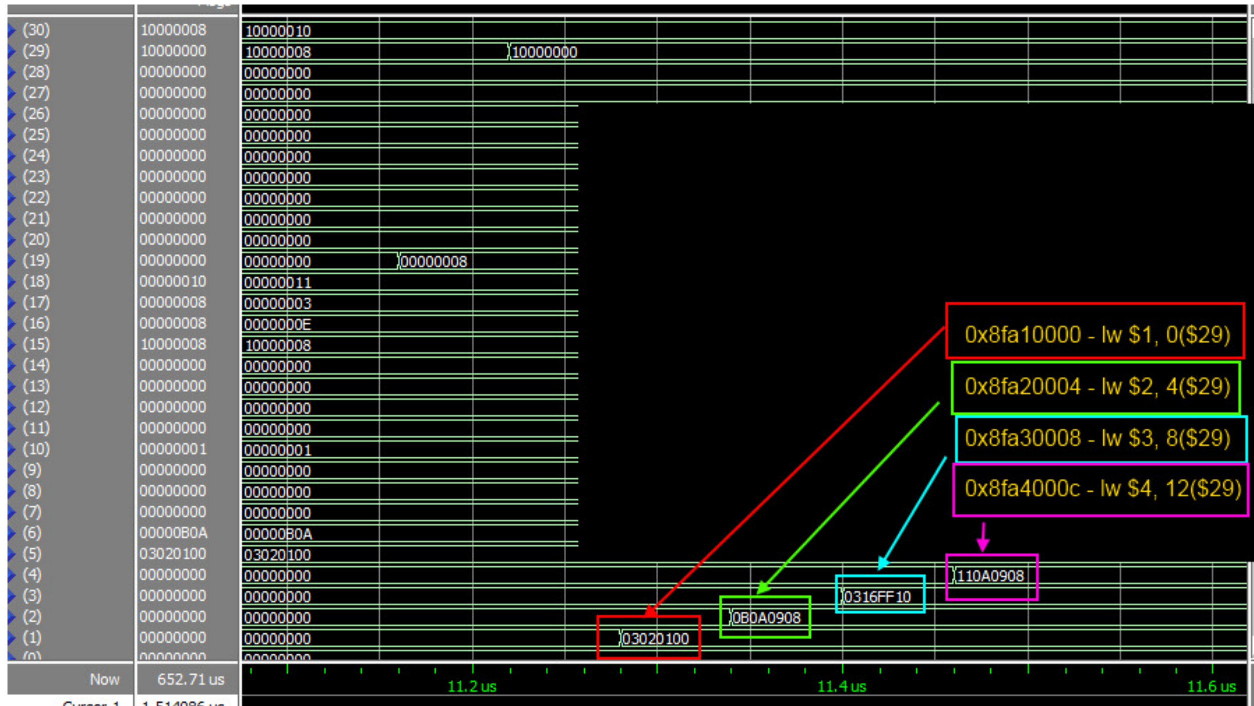


Figure 19 - Values of final registers

The program ends its execution by going into an infinite loop at the very end. A jump instruction keeps the loop going forever. The hazard mif file, when working correctly, outputs the following values

- $1 = 0x03020100
- $2 = 0x0b0a0908
- $3 = 0x0316ff10
- $4 = 0x110a0908

Full Register output

Signal	Value	Waveform
inst_mem_out_sig	93B10004	3C050302, 34A50100, 34060908, 3C1D1000, AFA50000, A7A60004, 34060B0A, A7A60006, 3C1E1000, 37DE0008, 03C07820, 93B20000, 93B10004
ra	{00000000} {1	{00000000} {00000000} {00000000} {00000000} {00000000}... {00000000} ... {00000000} ... {00000000} ... {00000000} {00000000} {10000000} {00000... {00000000} {00000000} {10... {00000000}...
(31)	00000000	00000000
(30)	10000000	00000000, 10000000
(29)	10000000	00000000, 10000000
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	00000000
(17)	00000000	00000000
(16)	00000000	00000000
(15)	00000000	00000000
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000000
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000000, 00000908, 00000B0A
(5)	03020100	00000000, 03020000, 03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	93B10004, 02328020, 2A0A000B, 11400002, 02128820, 08100013, 2A2A000B, 11400002, 02309020, 08100018, A3D20000, 23DE0001
ra	{00000000} {1	{00000... {00000000} ... {00000000} {10000008} {10... {00000000} {10000008} {10... {00000000} ... {00000000} {10000008} {10000000} {00000000} {00000000} {00000000} {00000000} {0000...
(31)	00000000	00000000
(30)	10000000	10000000, 10000008
(29)	10000000	10000000
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	00000000
(17)	00000000	00000000, 00000008
(16)	00000000	00000000, 00000008
(15)	00000000	00000000, 10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000000, 00000001
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	23DE0..., 23BD0001, 17AFFFEF, 34130008, 03B3E823, 93B20000, 93B10004, 02328020, 2A0A000B, 11400002, 02128820
ra	{00000000} {1	{00... {00000000} {10000008} {10000000} {00000... {00000000} ... {00000000} {10000009} {10000001} {00000000} {00000000} {00000000} {0... {00000000} ... {00000000} {10000009} {10... {00...
(31)	00000000	00000000
(30)	10000000	10000008, 10000009
(29)	10000000	10000000, 10000001
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	000..., 00000010, 00000001
(17)	00000000	00000008, 00000009
(16)	00000000	00000008, 000...
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000001
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	02… 08100013 2A2A000B 11400002 02309020 08100018 02119022 A3D20000 23DE0001 23BD0001 17AFFFEF 34130008 03B3…
ra	{00000000} {1	{00000000} {10000009} {10000001} {00000… {00000000} {10000009} {10… {00000000} {10000009} {10000001} {00000000} {00000000} {00000000} {0… {00000000} {10000009} {10… {00000…
(31)	00000000	00000000
(30)	10000000	10000009 1000000A
(29)	10000000	10000001
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	00000001 FFFFFFFF
(17)	00000000	00000009 0000000B
(16)	00000000	0000000A
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000001 00000000
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	03B3E823 93B20000 93B10004 02328020 2A0A000B 11400002 02128820 08100013 02128822 2A2A000B
ra	{00000000} {1	{00000000}… {00000000} {1000000A} {10000002} {00000000} {00000000} {00000000} {0… {00000000} … {00000000} {1000000A} {10… {00000000} {1000000A} {10000002} {00000000} {00000000…
(31)	00000000	00000000
(30)	10000000	1000000A
(29)	10000000	10000001 10000002
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	FFFFFFFF 00000002
(17)	00000000	0000000B 0000000A
(16)	00000000	0000000A 0000000C
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000000
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	2A2A000B 11400002 02309020 08100018 A3D20000 23DE0001 23BD0001 17AFFFEF 34130008 03B3E823 93B20000 93B10004
ra	{00000000} {1	{00000000} {1000000A} {10000002} {00000000} {0000… {00000000} {1000000A} {10… {00000000} {1000000A} {10000002} {00000… {00000000} … {00000000} {1000000B} {10000003} {00000000…
(31)	00000000	00000000
(30)	10000000	1000000A 1000000B
(29)	10000000	10000002 10000003
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	00000002 00000016
(17)	00000000	0000000A
(16)	00000000	0000000C
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000000 00000001
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	93B10... → 02328020 → 2A0A000B → 11400002 → 02128820 → 08100013 → 02128822 → 2A2A000B → 11400002 → 02309020 → 08100018 → 0...
ra	{00000000} {1	{00000000} {1000000B} {10000003... → {00000000} ... → {00000000} {1000000B} {10... → {00000000} ... → {00000000} {1000000B} {10000003} {00000000} {00000000} {00000000} {00000000} {00000000} {0...
(31)	00000000	00000000
(30)	10000000	1000000B
(29)	10000000	10000003
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	00000016 → 00000003
(17)	00000000	0000000A → 0000000B
(16)	00000000	0000000C → 0000000E
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000001 → 00000000
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	0... → 02119022 → A3D20000 → 23DE0001 → 23BD0001 → 17AFFFEF → 34130008 → 03B3E823 → 93B20000 → 93B10004 → 02328020
ra	{00000000} {1	{00000000} {1000000B} {10000003} {00000000} {00000000} {00000000} {00000000} {00000000} {00000000} {...} → {00000000} ... → {00000000} {1000000C} {10000004} {00000000} {00000000} {00000000...
(31)	00000000	00000000
(30)	10000000	1000000B → 1000000C
(29)	10000000	10000003 → 10000004
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	00000003
(17)	00000000	0000000B
(16)	00000000	0000000E
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000000
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	02328020 → 2A0A000B → 11400002 → 02128820 → 08100013 → 02128822 → 2A2A000B → 11400002 → 02309020 → 08100018 → 02119022
ra	{00000000} {1	{00000000... → {00000000} ... → {00000000} {1000000C} {10... → {00000000} {1000000C} {10000004} {00000000} {00000000} {00000000} {00000000} {00000000} {00000000} {00000000} {00000000} {0000...
(31)	00000000	00000000
(30)	10000000	1000000C
(29)	10000000	10000004
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	00000003 → 00000008
(17)	00000000	0000000B → 00000010
(16)	00000000	0000000E → 00000018
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000000
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	02119022, A3D20000, 23DE0001, 23BD0001, 17AFFFEF, 34130008, 03B3E823, 93B20000, 93B10004, 02328020, 2A0A000B
ra	{00000000} {1...	
(31)	00000000	00000000
(30)	10000000	1000000C, 1000000D
(29)	10000000	10000004, 10000005
(28)–(19)	00000000	00000000
(18)	00000000	00000008, 00000009
(17)	00000000	00000010
(16)	00000000	00000018
(15)	00000000	10000008
(14)–(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)–(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	2A0A..., 11400002, 02128820, 08100013, 02128822, 2A2A000B, 11400002, 02309020, 08100018, 02119022, A3D20000, 23...
ra	{00000000} {1...	
(31)	00000000	00000000
(30)	10000000	1000000D
(29)	10000000	10000005
(28)–(19)	00000000	00000000
(18)	00000000	00000009
(17)	00000000	00..., 000000FF
(16)	00000000	00000018, 00000108
(15)	00000000	10000008
(14)–(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)–(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	A..., 23DE0001, 23BD0001, 17AFFFEF, 34130008, 03B3E823, 93B20000, 93B10004, 02328020, 2A0A000B, 11400002, 02128...
ra	{00000000} {1...	
(31)	00000000	00000000
(30)	10000000	1000000D, 1000000E
(29)	10000000	10000005, 10000006
(28)–(19)	00000000	00000000
(18)	00000000	00000009, 0000000A
(17)	00000000	000000FF, 00000016
(16)	00000000	00000108
(15)	00000000	10000008
(14)–(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)–(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	02128820 · 08100013 · 02128822 · 2A2A000B · 11400002 · 02309020 · 08100018 · 02119022 · A3D20000 · 23DE0001 · 23BD0001
ra	{00000000} {1	{0000000... {00000000} {1000000E} {10000006} {00000000} {00000000} {00000000} {00000000} {00000000} {00000000} {00000000} {00000000} {00000000} {00000000} {0000000A} {00000016} {000...
(31)	00000000	00000000
(30)	10000000	1000000E
(29)	10000000	10000006
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	0000000A
(17)	00000000	00000016
(16)	00000000	00000108 · 00000020
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000000
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	23BD0001 · 17AFFFEF · 34130008 · 03B3E823 · 93B20000 · 93B10004 · 02328020 · 2A0A000B · 11400002 · 02128820 · 08100013
ra	{00000000} {1	{00000000} {1000000E} {10000006} {... {00000000} ... {00000000} {1000000F} {10000007} {00000000} {00000000} {00000000} {0... {00000000} ... {00000000} {1000000F} {10... {00000000} {100...
(31)	00000000	00000000
(30)	10000000	1000000E · 1000000F
(29)	10000000	10000006 · 10000007
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	0000000A · 0000000B
(17)	00000000	00000016 · 00000003
(16)	00000000	00000020 · 0000000E
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000000
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	0810... · 02128822 · 2A2A000B · 11400002 · 02309020 · 08100018 · A3D20000 · 23DE0001 · 23BD0001 · 17AFFFEF · 34130008 · 03B3E823 · 8F...
ra	{00000000} {1	{00000000} {1000000F} {10000007} {00000000} {00000000} {00000000} {00000000} {00000000} ... {00000000} {1000000F} {10... {00000000} {1000000F} {10000007} {00000... {00000000} ... {0000...
(31)	00000000	00000000
(30)	10000000	1000000F · 10000010
(29)	10000000	10000007 · 10000...
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000
(18)	00000000	0000000B · 00000011
(17)	00000000	00000003
(16)	00000000	0000000E
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000000 · 00000001
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000
(3)	00000000	00000000
(2)	00000000	00000000
(1)	00000000	00000000
(0)	00000000	00000000

Signal	Value	Waveform
inst_mem_out_sig	93B10004	8FA10000 8FA20004 8FA30008 8FA4000C 08100022
ra	{00000000} {1	{00000000} {10000010} {10... {00000000} ... {00000000} ... {00000000} ... {00000000} ... {00000000} ... {00000000} {10000010} {10000000} {0
(31)	00000000	00000000
(30)	10000000	10000010
(29)	10000000	10000008 10000000
(28)	00000000	00000000
(27)	00000000	00000000
(26)	00000000	00000000
(25)	00000000	00000000
(24)	00000000	00000000
(23)	00000000	00000000
(22)	00000000	00000000
(21)	00000000	00000000
(20)	00000000	00000000
(19)	00000000	00000000 00000008
(18)	00000000	00000011
(17)	00000000	00000003
(16)	00000000	0000000E
(15)	00000000	10000008
(14)	00000000	00000000
(13)	00000000	00000000
(12)	00000000	00000000
(11)	00000000	00000000
(10)	00000000	00000001
(9)	00000000	00000000
(8)	00000000	00000000
(7)	00000000	00000000
(6)	00000B0A	00000B0A
(5)	03020100	03020100
(4)	00000000	00000000 110A0908
(3)	00000000	00000000 0316FF10
(2)	00000000	00000000 0B0A0908
(1)	00000000	00000000 03020100
(0)	00000000	00000000
Now	652.71 us	