# 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	в	JAL
R	1a: exmen.nd=idex.rs(1) =idex.rt(2) 2a: memvb.nd=idex.rs(3) =idex.rt(4) 3a: memvb.nd=id(a,rs(5) =ijid.rt(6)	1a: exmem.nd=idex.rs(1) 2a: memwb.nd=idex.rs(3) 3a: memwb.nd=ifid.rs(5)	1a: exmem.nd=idex.rs(1) 2a: memwb.nd=idex.rs(3) 3a: memwb.nd=ifid.rs(5)	1a: exmen.ut=idex.cs(1) =idex.ct(2) 2a: memwb.ut=idex.cs(3) =idex.ct(4) 3a: memwb.ut=id(x;s(5) =idd.ct(6)	1a: idax.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: exmen.rd=ifid.rs(8) =ifid.rt(10) 3a: mermeb.rd=ifid.rs(5) =ifid.rt(6)	
AI	1a: exmem.d=idex.cs(1) =idex.d(2) 2a: memwk.d=idex.cs(3) =idex.d(4) 3a: memwk.d=idex.cs(5) =ide.d(6)	1a: exmem.f=idex.cs(1) 2a: memwb.f=idex.cs(3) 3a: memwb.f=ifd.cs(5)	1a: exment.1=idex.(5(1) 2a: memvb.(1=idex.(5(3) 3a: memvb.(1=idd.(5(5)	1a: exmem.d=idex.cs(1) =idex.d(2) 2a: memwb.d=idex.cs(3) =idex.d(4) 3a: memwb.d=idex.cs(5) =ifid.d(6)	1a: idex.tt=ifid.rs(stall) 2a: exmem.tt=ifid.rs(8) 3a: memmb.tt=ifid.rs(5)		1a: idex.tt=fid.rs(stall) =(fid.rt(stall) 2a: exmen.tt=fid.rs(8) =(fid.rt(10) 3a: mermet.tt=fid.rs(5) =(fid.rt(6)	
L	1a: ifid.,t=g(s,(stall) =g(s,,t(stall) 2a: memvkb,t=idex,(s(3) =idex,t(4) 3a: memvkb,t=idex,(s(5) =ifid.,t(6)	1a: i <u>fd_rt=pre.rs</u> (stall) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=i <u>fd.rs</u> (5)	1a: idd.rt=gre.rs(stall) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=idd.rs(5)	1a: (fd, d=gre, (s(stall) =gre, d(stall) 2a: mernyb, d=idex, (s(3) =idex, d(4) 3a: mernyb, d=ide, (s(5) =(fd, d(6)	1a: ifid.t=pre.cs(stall) 2a: idex.t=pre.cs(stall) 3a: memwb.t=ifid.cs(5)		1a: ifid, f=gre, rs(stall) =gre, ft(stall) 2a: idex, f=gre, rs(stall) =gre, ft(stall) 3a: mermeb, ft=ifid, rs(5) =ifid, ft(6)	
s								
JR								
LUI	1a: exmen.d=idex.cs(11) =idex.ct(12) 2a: menwb.d=idex.cs(13) =idex.ct(14) 3a: menwb.d=ide.rs(15) =fid.dt(16)	1a: exmem.d=idex.cs(11) 2a: memwb.d=idex.cs(13) 3a: memwb.d=idd.cs(15)	1a: exmem.d=idex.cs(11) 2a: memwb.d=idex.cs(13) 3a: memwb.d=ifd.cs(15)	1a: exmem.t=idex.ts(11) =idex.tt(12) 2a: memub.t=idex.ts(13) =idex.tt(14) 3a: memub.t=idd.ts(15) =fid.tt(16)	1a:idex.d=ifid.cs(17) 2a:excec.d=ifid.cs(18) 3a: <u>merowb.d=ifid.cs</u> (15)		1a: idex.t=ifid.s(17) =ifid.s(19) 2a: exmem.t=ifid.ts(18) =ifid.ts(20) 3a: memwb.t=ifid.ts(15) =ifid.t(16)	
в								
JAL	1a: exmem.31=idex.(5(21) =idex.(1(22) 2a: memwb.31=idex.(3(23) =idex.(1(24) 3a: memwb.31=ifd.(5(25) =ifd.(1(25)	1a:exmem.31=idex.ms(21) 2a:memwb.31=idex.ms(23) 3a:memwb.31=ifd(.ms(25)	1a:exmem.31=idex.(s(21) 2a:memwb.31=idex.(s(23) 3a:memwb.31=ifd(,(s(25)	1 a: exmem.31=idex.(s(21) =idex.(t(22) 2a: memwb.31=idex.(s(23) =idex.(t(24) 3a: memwb.31=ifd.(s(25) =ifd.(t(25)	1a: idex.31=ifid.rs(27) 2a: exmem.31=ifid.rs(28) 3a: memwb.31=ifid.rs(25)		1a: idex.31=(fid,rs(27) =(fid,rt(29) 2a: exmem.31=(fid,rs(28) =(fid,rt(30) 3a: memwb.31=(fid,rs(25) =(fid,rt(26)	

Figure 1 - Hazard table

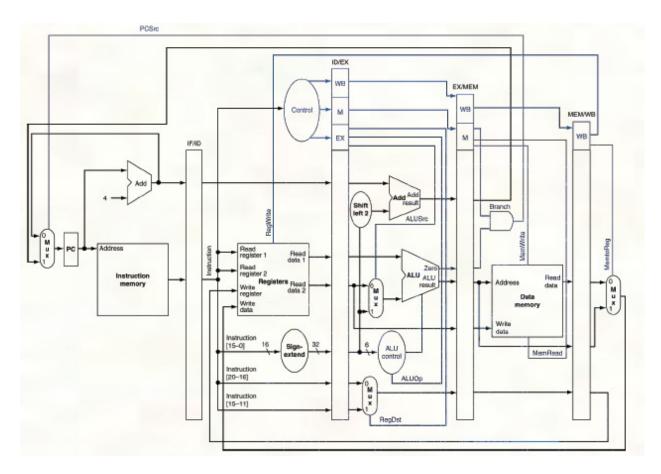


Figure 2 - Simplified view of the pipelined datapath and stages

# **Book Questions**

Using the blue book.

# <u>4.12.1</u>

a) Pipelined = 500ps, non-pipelined = (500+100+350+400+300) = 1650ps

b) Pipelined = 200ps, non-pipelined = (200+150+120+190+140) = 800ps

# <u>4.12.2</u>

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

# <u>4.12.3</u>

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

#### <u>4.20.1</u>

а	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)
b	add followed by sw (1 apart)
	add followed by lw (2 apart)
	sw followed by lw (1 apart)
	sw followed by add ( 2 apart)

#### <u>4.20.2</u>

· · · · · · · · · · · · · · · · · · ·	1
а	With forwarding
	o none
	Without forwarding
	<ul> <li>Iw followed by add (2)</li> </ul>
	<ul> <li>add followed by add (1)</li> </ul>
	<ul> <li>add followed by sw (2)</li> </ul>
	<ul> <li>add followed by sw (1)</li> </ul>
b	With
	$\circ$ Iw followed by add (1)
	Without forwarding
	$\circ$ add followed by sw (1)
	<ul> <li>Iw followed by add (1)</li> </ul>

# <u>4.24.1</u>

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

<u>4.24.2</u>

a) 0%

b) 25%

<u>2.24.3</u>

a) 75%

b) 60%

# Section 3: Instruction

<u>JR</u>

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.

Instruction=0x00 jr reg7	0E00008		uction= ess=0x \		0000308
⊢� address	19	19	1A		)C2
}� pc_in	00000068	0000068	(0000)	00000	308 (0000
⊢� pc_add_out	00000068	00000068	0000006	5C	0000030C
⊢� inst_mem_out	00E00008	00E00008	08100	00E0	3C1E1000
🔶 stall	0				
I mclk	1				
⊢� pc_out	00000064	00000064	0000006	58	00000308
⊢� inst_mem_out_sig	00E00008	00E00008	08100	00E0	(3C1E1000
– TE/TO					
z 🕤 Now	146.91 us				6.4u
Cursor 1	6.294327 us	6.294	327 us		

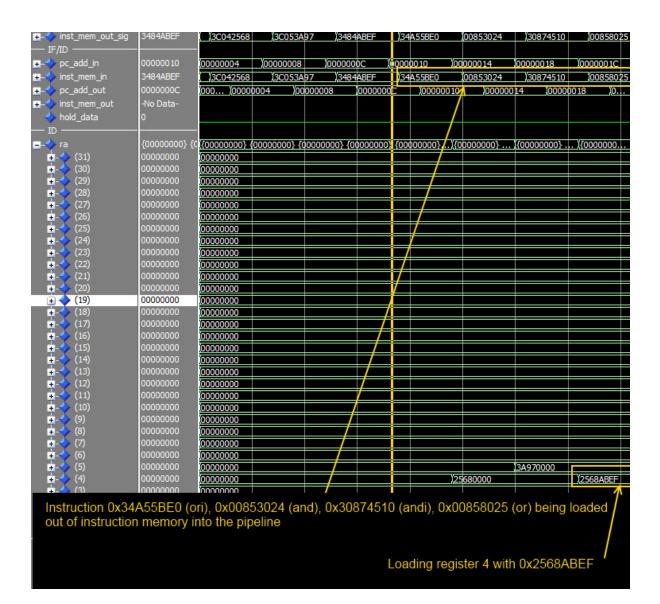
Figure 3 - JR going to contents of register 7

(12) (11) (10) (9)	Register 7 = 0x00000308 Forwarding is used in this example to get reg7 data over to the JR mux input								
(8)	00000000	0000000			<u> </u>				
(7)	00000000	0000000			00000308				
(6)	59E127AB	59E127AB							
🔶 (5)	59E0E367	59E0E367							
🔶 (4)	56784BCD	56784BCD							
Now	146.91 us	1.1	1	6.4	us				
Cursor 1	6.294327 us	6.2943	327 us						

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
s11	\$s2,\$a0,0x4
srl	\$s3,\$a0,0x2



**Figure 5 - Logical function test** 

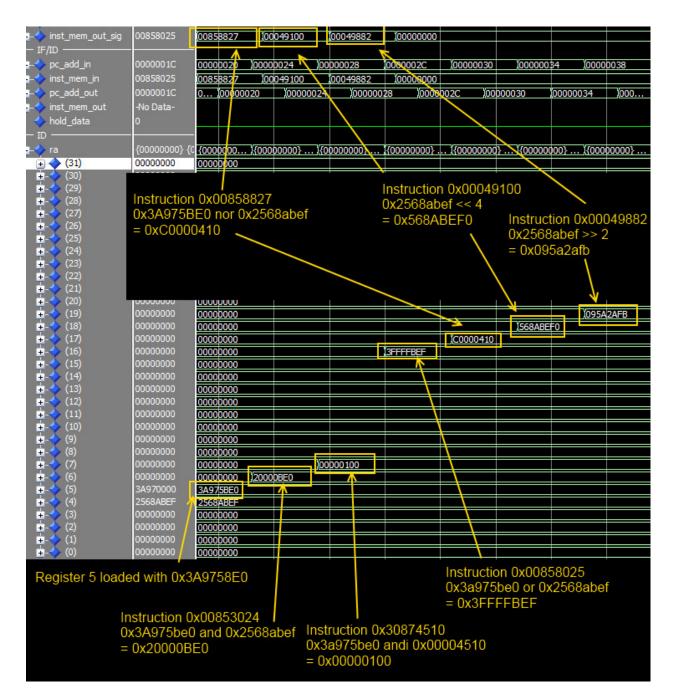
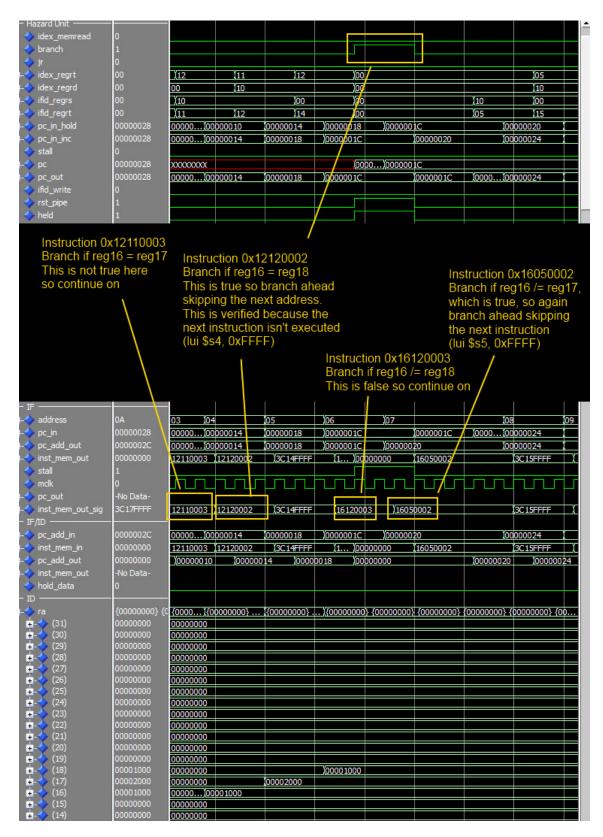
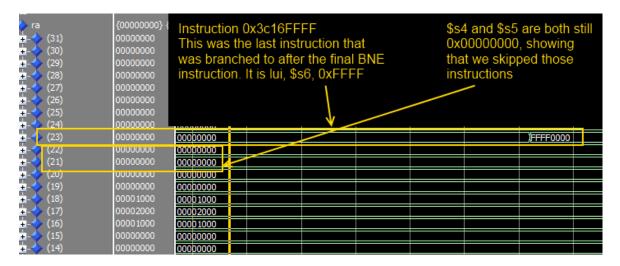


Figure 6 - Logical function test

# **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					





#### Figure 8 - Branch testing outcome

# SLT, SLTU, SLTI, SLTIU

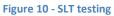
		SLT/SLTU/SLTI/SLTIU Test MIF	
lui	\$a0,0xFF11		
lui	\$s0,0xFFFF		
ori	\$s0,\$s0,0xFFFF		
ori	\$s1,\$s1,0x1234		
ori	\$s2,\$s2,0x2345		
slt	\$t0,\$s1,\$s2		
slt	\$t1,\$s2,\$s1		
sltu	\$t3,\$s0,\$s1		
sltu	\$t4,\$s1,\$a0		
slti	\$t5,\$s1,0x4000		
slti	\$t6,\$s2,0x1900		
sltiu	\$t7,\$a0,-6		
sltiu	\$t8,\$a0,0x4161		

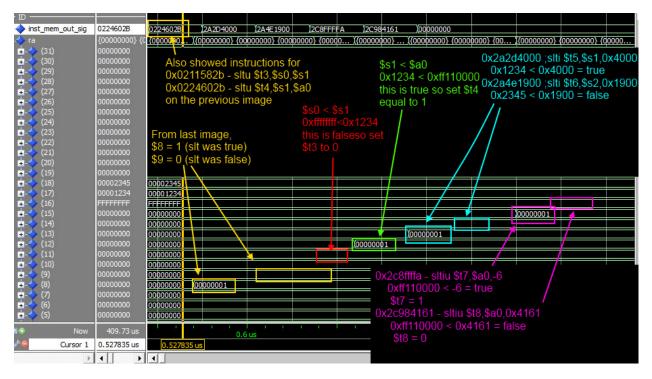
The difference between SLT and SLTU as well as SLTI and SLTIU is how the code perceives the data. Because my main example images didn't show the differences well enough I created another short MIF file to test the signed and unsigned version against two identical numbers. This can be seen and explained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

(21) (20) (19) (18)	00000000 00000001	00000000 00000000 00000000 00000000							00000	0001	
(18) (17) (16) (15)		000000000000000000000000000000000000000			FFFF0000	)FFFFFFFFF	000012	34			
\$s0 = (	(15)0000000000000000\$s0 = 0xFFFFFFF0x0211982a - slt \$s3,\$s0,\$s1 This checks is the signed version of 0xFFFFFFFF < 0x00001234 -1 < 4660, which is true so set \$s3, or \$reg19 = 10x0211a02b - sltu \$s4,\$s0,\$s1 This checks the unsigned version of the same numbers 0xFFFFFFF0x0211a02b - sltu \$s4,\$s0,\$s1 This checks the unsigned version of the same numbers 0xFFFFFFF										

Figure 9 - Signed vs Unsigned

· ID	0224602B	3C04FF11 (3C10FFFF )3610FFFF )36311234 36522345 )023	32402A )	0251482A	0211582B	0224602B
-🔷 ra	{00000000} {0	Q {{000006.0} {0000000} {0000000000000000000000000	0000} [{0000	0}/	0000000} ){0	0000000})
±> (31)	00000000					
<u> </u> <u> </u> <u> </u> <u> </u> (30)	00000000					
🛓 - 🔶 (29)	00000000					
🛓 - 🔶 (28)	00000000	Set \$a0 = 0xF1110000				
🛓 - 🔶 (27)	00000000	Set \$s0 = 0xFFFFFFF				
😐-🔷 (26)	00000000	Set \$s1 = 0x00001234				
÷> (25)	0000000	Set \$s2 = 0x00002345				
<b>⊕</b> -� (24)	00000000					
<u> </u> (23) (2) (	00000000					
<b>+</b> - <b>&gt;</b> (22)	00000000					
<b>±</b> - <b>(</b> 21)	0000000					
· <b>⊥</b> - <b>◇</b> (20)	00000000					
<b>+</b> - <b>(</b> 19)	00000000					
<b>+</b> - <b>(</b> 18)	00002345 00001234					0002345
+	and the second				001234	
+	FFFFFFF 00000000	0x0232402a - slt \$t0,\$s1,\$s2		FFFFFF		
+	00000000	\$t0=1 if \$s1<\$s2				
+	00000000					
+ (13)	00000000	0x1234 < 0x2345 is true				
+ (12)	00000000	s0 \$t0 gets set to 0x1 in				
+ (11)	00000000	the next image				
+	00000000					
+	00000000	0x0251482a - sit \$t1,\$s2,\$s1				
+	00000000					
÷	00000000	\$t1=1 if \$s2 < \$s1				
÷	00000000	0x2345 < 0x1234 is false				<b>T</b>
<b>—</b>		so \$t1 stays at 0x0000000	1 1			
Now	409.73 us			0.4 us		
2 Cursor 1	0.527835.us					0.527835.00







# 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				

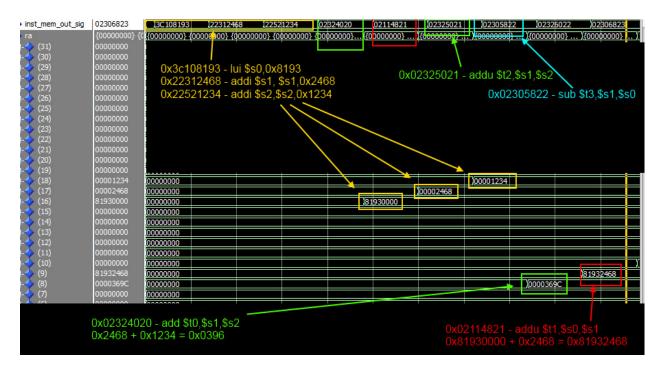
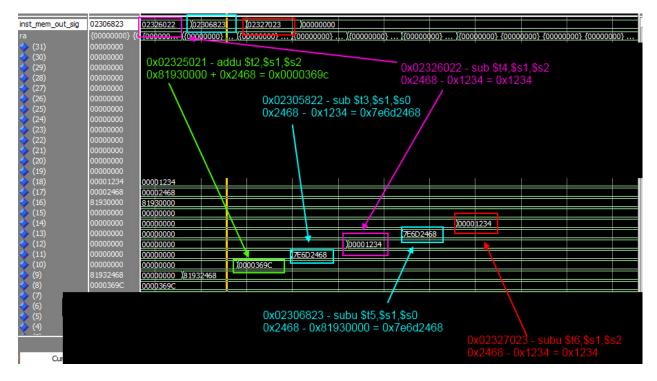


Figure 12 - SUB/SUBU/ADD/ADDU





SW, SH, SB

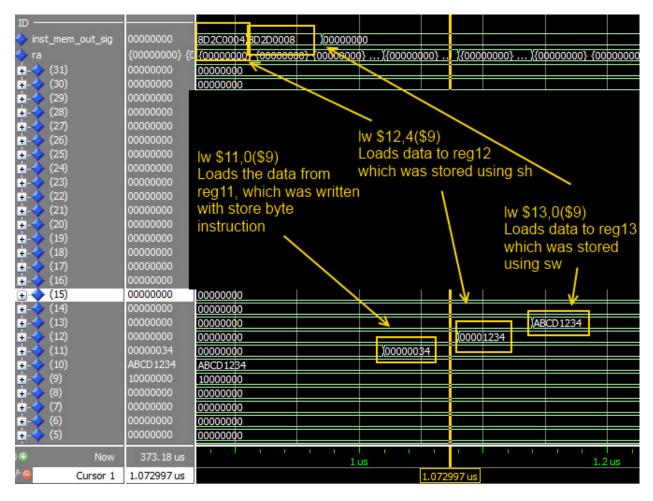
Test Code MIF						
lui	\$9,0x1000					
lui	\$15,0x0000					
ori	\$15,\$15,0x0000					
SW	\$15,0(\$9)					
SW	\$15,4(\$9)					
SW	\$15,8(\$9)					
lui	\$10,0xABCD					
ori	\$10,\$10,0x1234					
lui	\$2,0xffff					
lui	\$3,0xffff					
sb	\$10,0(\$9)					
sh	\$10,4(\$9)					
SW	\$10,8(\$9)					
lw	\$11,0(\$9)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

ID											
inst_mem_out_sig	3C03FFFF	3C091000	(3C0F0000	35EF0000	AD2F0000	AD2F0004	AD2F0008	J3C0AAB	CD 354A	1234 30	02FFFF
🔶 ra	{00000000} {0	{( <mark>0000000)} {</mark> 0	000000003 {0000	0000} {0000000}	{00000000}	{opooooda} {o	0000000} {000	00000} {0000	0000} {0000	000000} {000	00} {00
🛓 - 🔶 (31)	00000000	00000000	- <u> </u> -							1	
🛓 - 🔷 (30)	00000000	)	1			w\$15,0(\$9	N .		lui\$10.0	DxABCD	
🛓 - 🔷 (29)	00000000	i lui\$9,0x	1000								2.4
🛓 - 🔷 (28)	00000000	lui\$15.0	x0000			sw\$15,4(\$9				\$10,0x12	54
🛓 - 🔷 (27)	00000000		15,0x0000		5	sw\$15,8(\$9	)		lui\$2,0>	cffff	
🛓 - 🔷 (26)	00000000	011010,0	13,02000		These a	are used to	allocate t	he			
🛓 - 🔷 (25)	00000000	)				space so					
🛓 - 🔷 (24)	00000000	2									
🛓 - 🔶 (23)	0000000					w on addre		ote			
🛓 - 🔶 (22)	00000000	2			to using	sh/sb/sw.	Store all				
🛓 - 🔶 (21)	0000000	lui \$9_0	0x1000 take	s	0's to th	ese addres	SS				
😐 - 🔶 (20)	0000000		lete cycles								
<b>+</b> - <b>&gt;</b> (19)	00000000										
<b>+</b> - <b>&gt;</b> (18)	0000000		en back to t								
<b>+</b> - <b>&gt;</b> (17)	00000000	register	file and sto	ored							
<b>∔</b> - <b>◇</b> (16)	0000000										
<b>+</b> - <b>→</b> (15)	0000000										
<b>+</b> - <b>◇</b> (14)	00000000	0000000									
+	0000000	00000000									
±-◆ (12)	0000000	0000000									
±-◆ (11)	0000000	0000000				X					
±-◆ (10)	ABCD0000	0000000									
<u>+</u> -∲ (9)	1000000	00000000				10000000					
±- <b>☆</b> (8)	00000000	0000000									
±- <b>↑</b> (7)	00000000	00000000									
±- <b>↑</b> (6)	00000000	0000000									
<b>‡</b> -∲ (5)	0000000	00000000									

Figure 14 - Setting up the registers for the store test

– ID −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−	3C03FFFF	03FFFF A12A0000 XA52A0004 XAD240008 X8D280000 X8D2C0004 X8D2D0008 X0000000
		<u></u>
÷- (31)		
	00000000	Store byte
	00000000	sb \$10, 0(\$9)
I I LL	00000000	
+	00000000	V Store the byte of reg10
+		0.0xABCD at the bottom byte of Store word
+		
+ (23)		Store whole word in reg10
+	LOOOOOOC From	previous image, we load Store halfword
(22)	00000000 0xABC	CD1234 into reg10 to sh \$10,4(\$9) Load word
+		sthe sh (sh /sw) Ebdd Word
+	00000000 instruc	Store the nativord of region in err,s(eo)
+	00000000	to address 0x10000001
+	00000000	(because of offset=4)
+	00000000	
+	00000000	
+	0000000 00000	
+	0000000 00000	0000
+	00000 00000	
+	00000000 00000	
÷	000000000000000000000000000000000000000	0000 ABCD0000 ABCD1234
🛓 - 🔶 (9)	10000000 10000	
🛓 - 🔶 (8)		
🛓 - 🧇 (7)	0000000 00000	
🛓 - 🔶 (6)	0000000 00000	
🛓	0000000 00000	
Now	373.18 us	
		0.8 us 1 us
Cursor 1	0.587389 us	0.587389 us

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





# <u>LW, LHU, LBU</u>

	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)					

ID	36520000	13C101000 3C08BAD2 13508BEEF 136520000 14608000 18609000 1960A0000 1920B0000							
ra									
1 A rea	000000000000000000000000000000000000000								
+-~ (31) +-~ (30)	00000000								
+	00000000								
+	00000000								
+	00000000								
+	00000000	0xae080000 = sw \$t0,0(\$s0)							
+	00000000	$0_{\rm M}$							
÷	00000000								
÷	00000000	3508beef;ori\$t0,\$t0,0xbeef \ 0x90080000 = Intu \$t2,0(\$s0)							
<del>+</del> - <del>(</del> 22)	00000000	4 cycles later we write the data 0x920b0000 = lbu \$t3,0(\$s0)							
+	00000000	to the register file							
<b>+</b>	00000000								
<b>+</b> - <b>(</b> 19)	00000000	Ī. laita kara kara kara kara kara kara kara ka							
<b>±</b> - <b>〈</b> (18)	0000000	Ī.							
÷	0000000								
🛓 - 🔶 (16)	10000000								
🛓 - 🔶 (15)	0000000	00000000							
+ 🔶 (14)	00000000								
😐 - 🔶 (13)	0000000								
±	0000000								
±									
±	A DECEMPEND OF COLORS								
±-→ (9) +-→ (8)	0000000								
	00000000	00000000 BAD28EEF							
	00000000								
+- <b>&gt;</b> (6) +- <b>&gt;</b> (5)	00000000								
H-A (2)	0000000								
Now	441.54 us	00 us 0.2 us 0.4 us							
Cursor 1	0.283523 us	0.283523 us							
	• •								

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

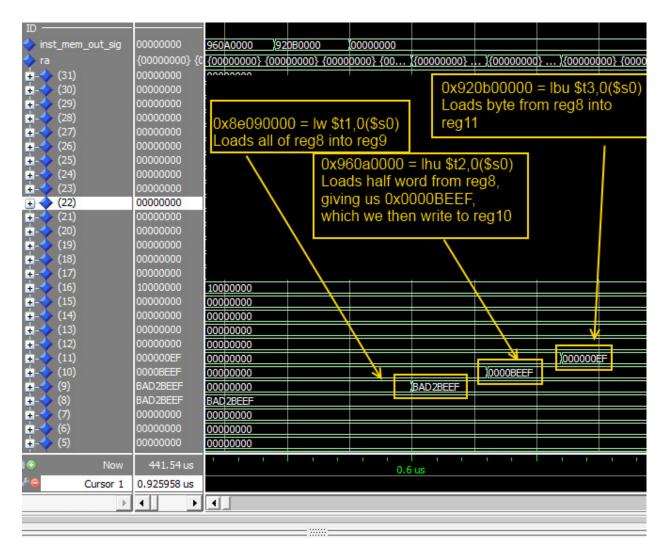


Figure 18 - Register values from the result of LW, LHU, and LBU

# 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				
	00000000	03020100 00000000	XXXXXX 10 000000 10	
	08	01 10 108	09	
-🔶 instr	08100018	11 )02309020 )08100018	A3D20000 23DE0001	
-🔷 i_regdata	00000008	00 00000008	100000000 10000008	
-🔷 off	0	2 )0	<u> </u>	
-🔷 o_data_mem	00000000	03020100 00000000	XXXXXX 10 (0000010	When store t = 1 we use the
-🔶 o_regdata	0000008	00)00000008	<u>)000000010 (10000008</u>	fwd_data value for the data
-🔷 o_address	02	00 04 02		
-🔷 byte_en	F	F. Contraction of the second s	<u>(1)</u>	value to be written to
-🔷 opcode	02	04 )00 )02	)28 )08	memory
-🔷 wb_data	00000010	00000001 00000010	00000008 (10000008	
-🔷 wb_regrd	12	0A )00 )12	<u>110 112</u>	
-🔶 fwd_data	00000001	00	<u>)/00000010 )/(0000008</u>	
store_t	0			
- Data Mem				
	02	00 04 02		
-🔶 byteena	F	F.	<u>χ1</u>	wren is high, data going in is
-🔶 data	00000008	00 (00000008	00000010 (10000008	0x10, which is SB
i 🔶 wren	0			oxro, which is ob
-🔶 q	00000000	03020100 (00000000	XXXXXX 10 000000 10	Reg18 = 0x00000010
-🔶 sub_wire0	00000000	03020100 00000000	XXXXXX 10 0000010	Kegre execcedere
- Fwd Mux B				
-🔶 in0	00000000	00000008 00000000	00000001	
-🔶 in1	00000010	00000001 00000010	20000008 21000008	
-🔶 in2	0000008	00 (00000010 )00000008	<u>)10000008 )10000009</u>	
-🔶 in3	00000000	0000000		
≅.⊛ Now	652.71 us	1.4 us	1.6 us	

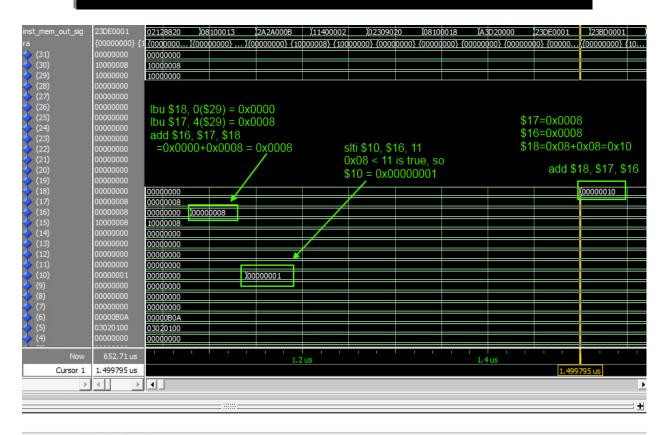
-🔷 (22)	00000000	0000000
-🔷 (21)	00000000	0000000
I-🔷 (20)	00000000	0000000
	00000000	0000000
	00000010	00000000 000000000000000000000000000000
-🔷 (1/)	ບບບບບບຮ	0000008
-🔷 (16)	0000008	
-🔷 (15)	10000008	Reg18 contains 0x00000010,
-🔷 (14)	00000000	this is used with SB to the
-🔷 (13)	00000000	first byte, so we write 0x10
-🔷 (12)	00000000	
	00000000	to the bottom byte of that address
-🔶 (10)	00000001	0

- Forwarding						
🔶 jump	0					
🛶 ex_regrs	1E	1D	00		1E	
	1E	06	(1E			)00
- i mem_regrd	1E	06			1E	
	06	06				(1E
🛶 alu_sela	2	0			2	
🛶 alu_selb	0	0				
🔷 regwrite	1					
🔶 wb_regwrite	0					
🔷 ex_instr	37DE0008	A7A60006	3C1E10	00	37DE00	8 )03
🔶 jr	0					
🔶 wb_jump	0					
🔶 itype	1					
🔶 store	0					
- IF						
🛶 address	0B	08 (09		0A		DB
🛶 pc_in	00000030	00 )000	00028	000000	2C	00000030
	00000030	00 (000	00028	000000	2C	00000030
🛶 inst_mem_out	03C07820	3C1E 3	37DE0008	03C0	7820	(93B200

# ex.rs = mem.rd so set forwarding mux A to forward mem.rd into the ALU

ID											
inst_mem_out_sig	03C07820	3C1E	37DE0008	03C07820		(93B2000	)0 ),93B	10004	02	328020	)
🔶 ra	{00000000} {0	{0000000	00} {0000	{00000000}	(00	000000} {1	0 ){0000	0000}	(000	00000}	
1 <b>(</b> 31)	00000000	0000000									
÷	00000000	0000000	þ		F		(10000	000	1000	0008	
÷ (29)	10000000	1000000	h				<u> </u>		>		
+	00000000	0x3c	1e0000 ·	lui \$30, 40	<u>)</u> 96	6 /					
<b>∔</b> -�� (27)	00000000	( 0v37	400008	ori \$30, \$	30	°. –					
🛓 🔶 (26)	00000000							41 A1		- 41	
🛓 🔶 (25)	00000000			0000000							IU –
🛨 - 🔷 (24)	00000000	( 0x8 (	goes into	the secor	nd	input a	nd we ge	et the o	corre	ect	
🛨 - 🔶 (23)	00000000	result from forwarding									
± (22)	0000000	7									

- regit	10	
– EX –––––		
🛶 in0	00000000	10000000 Хооооофо
🖂 in1	10000006	10000004 X00000B0A X10000006 X10000000 X
🖂 in2	10000000	00000B0A (10000006 (100000000 (10000008
🖂 in3	00000000	00000000
🛶 sel	2	
⊢� o	10000000	
⊢� in0	00000008	Forward mux A, which outputs into
🛶 in1	10000006	the top ALU input, is told to forward
🛶 in2	10000000	
🛶 in3	00000000	mem.rd by sel=2
👆 sel	0	So it outputs 0x10000000
- A 0	00000008	



Ita: 1 sim:/top\_level\_th

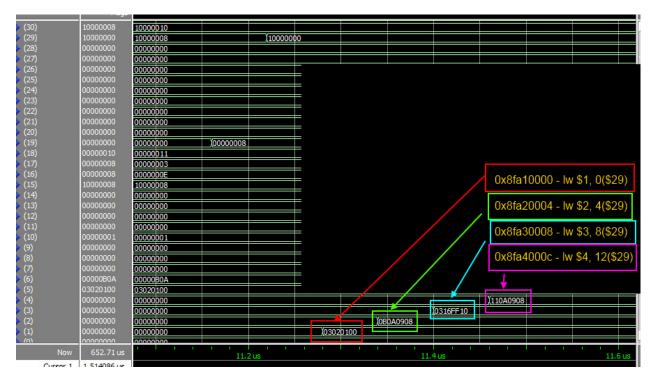


Figure 19 - Values of final registers

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

> inst_mem_out_sig 938100 ▶ ra {00000			Хатабооо4 Хзчобовфа Хатабоо Хсоооооооо3 Хсооооооод Хсоооооооо		
(31) 000000 (30) 100000	000 0000000 000 000 000 000 000 000 00				(1000000
- (29) 100000 - (28) 000000 - (27) 000000 - (26) 000000	000 0000000 000 000 000 000 000 000 00		(1000000) 		
	000 0000000				
	000 0000000 000				
	000 0000000				
	000 0000000 000 000 000 000 000 000 00				
	000 0000000				
	000 0000000				
	000 0000000 000 000 000 000 000 000 00				
	00000000000000000000000000000000000000	(03020000	)00000908 (03020100	00000E	
	000 0000000 000 000 000 000 000 000 00				
	000 0000000 000 0000000				
inst_mem_out_sig			)02128820 )08100013  2A2A000B 10008} {10 }{000000000} }{00000000} {1		
(31) 0000000 (30) 1000000	00 00000000000000000000000000000000000				
	00000000				
(26)         0000000           (25)         0000000           (24)         0000000           (23)         0000000	000000000000000000000000000000000000000				
(23) 000000 (22) 000000 (21) 000000	000000000000000000000000000000000000000				
(20) 0000000 (19) 0000000	00 00000000 00 0000000				
(17) 0000000 (16) 0000000	00 0000000 00 0000000	)0000008	X00000008		
- (15) 0000000 - (14) 0000000 - (13) 0000000	00 0000000 00	8000			
	00 00000000		χοοροοοο 1		
			,0000001		
(7)     0000000       (6)     0000000       (5)     0302010	03020100				
- (4) 0000000 - (3) 0000000 - (2) 0000000	000000000000000000000000000000000000000				
	00 0000000 00 0000000	u u			u and a second sec
inst_mem_out_sig 9381000 ra (000000 (31) 0000000 (30) 1000000	000} {1 <u>{00 }{00000000} {10000008} {</u> 00 00000000	10000000} {00000 <u>}</u> {00000000} <u>}</u> {0000	20000 )93810004 )( 0000} {10000009} {10000001} {00000000} {0		)11400002 )02128820 ) {00000000} {10000009} {10 }{00
(29) 1000000 (28) 0000000	00 10000000 00 000000000000000000000000	<u>)10000009</u> (10000	001		
(27) 0000000 (26) 0000000 (25) 0000000	00 00000000 00 00 00 00 00 00 00 00 00				
<ul> <li>(24)</li> <li>(23)</li> <li>(22)</li> <li>(2000000000000000000000000000000000000</li></ul>	00 0000000 00 00 00 00 00 00 00 00 00 0				
(21) 000000 (20) 0000000	00 00000000 00 0000000				
(19)       0000000         (18)       0000000         (17)       0000000	00 000 00000010 00 00000008			(0000000 1	0000009
(16) 0000000 (15) 0000000 (14) 0000000	00 0000008				)000
(13) 0000000 (12) 0000000 (11) 0000000					
(10) 0000000 (9) 0000000	00 00000001 00 0000000				
(16)         000000           (15)         000000           (14)         000000           (13)         000000           (11)         000000           (11)         000000           (10)         000000           (10)         000000           (10)         000000           (10)         000000           (10)         000000           (10)         000000           (10)         000000           (10)         000000           (10)         000000           (2)         0000000           (2)         0000000           (2)         0000000           (2)         0000000           (2)         0000000           (2)         0000000           (2)         0000000           (2)         0000000           (2)         0000000	00 00000000 DA 0000060A				
<ul> <li>(5)</li> <li>0302010</li> <li>(4)</li> <li>0000000</li> <li>(3)</li> <li>0000000</li> </ul>	00 00000000				
(2) (1) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0	00 00000000 00 00 00 00 00 00 00 00 00				
			v	U U	

inst_mem_out_sig	93B10004	oz %810b013 %24240008 \$11400002 %02309020 %081000\$8 %02119022 \$43020000 %230E0001 %23800001 %174+FFFF %34130008 \$03	
ra 	{00000000} {1 00000000	{100000000} {10000000} {10000000} {00000000} {1000000000} {1000000000} {1000000000} {10000000000	00
(30) 	10000000 10000000	1000009 (1000001 (10000000 (100000000	000A
-🔶 (28)	00000000	0000000	
	00000000		
	00000000 00000000		
	00000000 00000000		
	00000000 00000000		
- (20) - (19) - (18)	00000000	0000000	
	00000000 00000000	00000001 )FFFFFFF 00000009 )00000008 ) / 00000008 // 00000008 // 00000009 // 000000008 // 000000008 // 000000008 // 00000000	
	00000000 00000000	Ĵ0000000A [ ] _ ] ] _ ] ] _ ] ] _ ] ] _ ] ] _ ] ] _ ] ] _ ] ] _ ] ] _ ] _ ] ] ] ] ] ] ] ] ] ] ] _ ] ] ] ] _ ]	
	00000000		
	00000000		
-🔶 (10)	00000000	0000000( 000000 000000 000000 000000000	
	00000000 00000000		
	00000000 00000B0A	0000000 0000000 0000000 0000000 0000000	
	03020100 00000000		
	00000000	0000000	
	00000000		
	0000000		
			_
> inst_mem_out_sig > ra	93810004 {00000000} {:	103838823 /93820000 // 193810004 /02528020 // 2A0A0006 /11400002 /02128820 /08100013 /02128822 // 2A2A0 {00000000} {(00000000} {100000003 {00000000} {00000000} {00000000} {0 {(00000000} {100000004 {10 }(00000000} {100000004 {100000004 {100000004 {100000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {100000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {100000004 {1000000004 {1000000004 {1000000004 {1000000004 {1000000004 {100000004 {100000004 {100000004 {100000004 {100000004 {100000004 {100000004 {1000000004 {1000000004 {100000004 {100000004 {100000004 {100000004 {100000004 {100000004 {10000000000	
(31)	00000000	0000000 100000A 100000A	
(29) (28)	10000000 00000000		
	0000000	0000000	
	00000000 00000000	00000000 00000000000000000000000000000	
(25) (24) (23) (22) (22) (22) (22) (20) (20) (20) (20	00000000 00000000		
-2 (22) -2 (21)	00000000 00000000		
(20)	00000000	0000000	
	00000000	00000000  FFFFFFF  D0000002  FFFFFFF  D0000002  FFFFFFF  D0000002  FFFFFFF  D0000002  FFFFFFFF  D0000002  FFFFFFF  D0000002  FFFFFF  D0000002  FFFFFF  D0000002  FFFFF  FFF	
	00000000 00000000	0000000B 000000A 0000000A 000000C	
	00000000 00000000	10000008	
	00000000 00000000		
(11) (10)	0000000	0000000	
(10)	00000000		
-	00000000 00000000	0000000	
	00000B0A 03020100	00000E0A 08000E00 08000000	
	00000000 00000000		
(2) (1)	00000000 00000000		
	00000000		
<pre>inst_mem_out_sig ra</pre>	93810004 {00000000} {	2424b008 )1140b002 /02509020 /08100018 /43020000 )23060001 /2380b001 )17AFFEF [34130008 /03836823 )93820000 )93810004 /00000000} {10000004} {10000002} {00000000} {0000 {00000000} {100000004} {10000004} {10000004} {10000002} {00000 {00000000} {100000004} {10000003} {0000000}	00
+	00000000	00000000 10000000 10000000 10000000 1000000	
(29) (28)	10000000 00000000	10000002 Υμούουος 00000000 Υμούουος	
(27)	0000000	0000000	
(25)	00000000		
	00000000 00000000 00000000 00000000		
	00000000 00000000		
	00000000		
2 (25) 3 (24) 4 (23) 5 (23)	00000000	00000000 16 0000000 16 000000 16 0000000 16 00000000	
(16)	00000000		
(15)	0000000		
(13) (12)	00000000 00000000		
∎-◆ (11) ∎-◆ (10)	00000000	00000000 00000000 000000000000000000000	
(9)	00000000 00000000 00000000	00000000	
(7)	00000000		
	00000B0A 03020100	00000E0A	
∎- <b>→</b> (4) ∎- <b>→</b> (3)	00000000 00000000		
+	00000000		
	00000000		
			-

∍ inst_mem_out_sig → ra	93B10004 {00000000} {	93810[02328020] //2A0A0008 //11400002 /02128820 /08100013 //02128822 / /2A240008 //1400002 /02209020 /08100018 //0. [00000000] (10000008) (10000003
(31) (30) (29)	00000000 10000000 10000000	
-🔶 (28)	00000000	
	00000000	
-🔶 (21)	00000000 00000000 00000000	00000000000000000000000000000000000000
	00000000 00000000 00000000	00000000000000000000000000000000000000
	00000000 00000000 00000000	0000000A 000000B 000000C 000000B 000000C 000000C 0000000C 0000000C 000000
- (14) - (13) - (12)	00000000 00000000 00000000	
	00000000 00000000 00000000	00000000 0000000 00000000 000000000 0000
(16) (15) (14) (14) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (14) (13) (13) (14) (13) (14) (13) (13) (14) (13) (13) (14) (13) (14) (13) (14) (13) (14) (13) (14) (15) (14) (15) (14) (15) (14) (15) (14) (15) (14) (15) (	00000000 00000000 00000000	
	03020100 00000000	0000080A 0000000 000000 000000 000000000
	00000000 00000000 00000000	00000000000000000000000000000000000000
<b>⊦-</b> ◆ (0)	00000000	
inst_mem_out_sig ra 		0
- <b></b>	00000000 10000000 10000000	00000000 10000005 10000005
	00000000 00000000 00000000	00000000 00000000 00000000
	00000000 00000000 00000000 00000000	00000000 00000000 00000000
- (28) - (27) - (25) - (24) - (23) - (15) - (15) - (13) - (13)	00000000 00000000 00000000	00000000
	00000000 00000000 00000000	00000000 0000003 0000003
	00000000 00000000 00000000	000000E
	0000000 0000000	
	00000000 00000000 00000000	00000000 00000000 00000000
- (11) - (10) - (9) - (8) - (7) - (6) - (5) - (4) - (3)	00000000 00000000 00000B0A	00000000 00000000 00000000
	03020100 00000000 00000000	03020100 0000000 0000000 0000000 00000000 0000
- (2) - (1) - (0)	00000000 00000000 00000000	00000000 00000000 00000000 0000000 00000
inst_mem_out_sig ra 	0000000	0238820 2AAA0006 111400002 02128820 0010013 02128822 124240006 11400002 02309020 085100018 02119022 {00000000}{(000000002)
	10000000 10000000 00000000	1000000C 100000C 10000004 0000004 0000000 0000000 0000000 000000
	00000000 00000000 00000000 00000000 0000	00000000 00000000 00000000
- (24) - (23) - (22)	00000000 00000000 00000000	00000000 00000000 00000000
- (21) (20) (19)	00000000 00000000 00000000	0000000
	00000000 00000000	00000003 )00000008
	000000000000000000000000000000000000000	0000000E 00000018 00000018 000000018 0000000000
- (29) (29) (27) (25) (25) (25) (25) (22) (22) (22) (22	00000000 00000000 00000000 00000000 0000	00000000 00000000 00000000
	00000000	00000000 00000000 00000000
	00000000 00000B0A 03020100	00000000 0000000A 03020100
(4) (3) (2)	00000000 00000000 00000000	0000000
	00000000	

inst_mem_out_sig	93810004	02119022 (A3D20000 )23DE	0001 )238D0001 (17AF	FFEF (34130008 )0	3836823 (93820000	93B 10004	(02328020 )2A0A000B )
ra 	{00000000} {: 00000000						{00000000} {00000000} {0 <u>}</u> {00000000} <u>}</u> {
(30)	10000000 10000000	1000000C 10000004		χ100	0000D (10000005		
	00000000	0000000					
-2 (25) -2 (25) (24)	00000000 00000000 00000000	000000000000000000000000000000000000000					
- (29) - (28) - (27) - (26) - (25) - (24) - (23) - (23) - (22)	00000000	00000000					
	00000000	00000000					
	00000000	00000008					)0000000
- (17) - (16) (15)	00000000	00000010 00000018 10000008					
	00000000 00000000 00000000	10000008 00000000 00000000					
	00000000	00000000					
-🔶 (10)	00000000 00000000	00000000					
	00000000	0000000					
(5) (4)	00000B0A 03020100 00000000	00000BDA 030201D0 000000D0					
	00000000	00000000					
	00000000 00000000	00000000					
· * •							
inst_mem_out_sig ra		2A0A)11400002 02128820			000B 11400002 00230		2119022 [A3D20000 ]23 0000} {00000000} {00000000} {0000000} {000
(31)	0000000	00000000 1000000D					
- <b>-</b> (29) - <b>-</b> (28)	1000000 00000000	10000005 00000000					
	0000000	00000000					
- - - - - - - - - - - - - - - - - - -	0000000	00000000					
- (24) - (23) - (22) - (21) - (20) - (19) - (18)	0000000	00000000					
	0000000	00000000					
-🔶 (17)	00000000 00000000	00000009 00)000000FF					
(16) (15) (14) (13)	0000000	00000018 10000008	00000108				
	0000000	00000000					
(12) (11) (10)	0000000	00000000 00000000 00000000					
	0000000	00000000					
- (9) - (8) - (7) - (6) - (5) - (4)	00000000 00000B0A	00000000 00000B0A					
	0000000	03020100					
- (3) (2) (1)	0000000	00000000 00000000 00000000					
		00000000					
inst_mem_out_sig	93810004	A)23DE0001 )23BD0001	(17AFFFEF )34130008	0383E823 (93820000	)93B 1D00	4 )02328020	)2A0A000B (11400002 )02128
ra 	00000000	0000000			{1000000E} {10000006} {00000	0000} {00000000} {0000000}	{0 }{00000000} }{00000000} {1000000E}
(30)	10000000 10000000 00000000	1000000D 10000005		(1000000E (10000006			
-2 (28) -2 (27) -2 (26)	00000000	00000000 00000000 00000000					
(25)	00000000	00000000					
	00000000 00000000	0000000					
- <b>2</b> (21) - <b>2</b> (20)	00000000 00000000	0000000					
	00000000	0000000					0000000A
	00000000 00000000 00000000	000000FF 00000108 10000008					00000016
	00000000 00000000 00000000 00000000 0000	0000000					
	00000000	0000000					
	00000000	0000000					
	00000000 00000000 00000B0A	00000000 00000000 00000B0A					
	03020100 00000000	03020100 00000000					
- (29) (29) (27) (25) (25) (25) (22) (22) (22) (22) (22	00000000	0000000					
- <b>(</b> 1) - <b>(</b> 0)	00000000 00000000 00000000	0000000					

nst_mem_out_sig	93B10004	02128820	08100013	02128822		12A2A00	DB 111400	002 02	309020	08100018	02119022		JA3D	20000	23DE0	001 I23	BD0001
	{00000000} {:	{0000000 <u>}</u> {0															
(31)	00000000	00000000					01 1000000000								0000000		01 10001
(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															
(10)	00000000	00000016															
(16)	00000000		000020														
(15)	00000000	10000008	000020														
(13)	00000000	00000000															
(13)	00000000																
(13)	00000000	0000000															
(12)		0000000															
(11)	00000000	0000000															
(10)	00000000	0000000															
(9)		0000000	_	_		_											
(8)	00000000	0000000		_													
(7)	00000000	0000000															
(6)	00000B0A	00000B0A															
(5)	03020100	03020100															
(4)	00000000	0000000															
(3)	00000000																
		00000000															
🧇 (2)	00000000	00000000															
(2) (1)	00000000 00000000	00000000															
🧇 (2)	00000000	00000000						U	J.								
(2) (1) (0) inst_mem_out_sig	00000000 00000000 00000000 938 10004	00000000 00000000 00000000 238D0001 (174		+130008 )		)93820000		) <u>93810</u>		328020		)2A0A000B			)02128820		
(2) (1) (0) inst_mem_out_sig ra	00000000 00000000 00000000 93810004 {00000000} {	00000000 00000000 00000000 23800001 )(17#					000000F} {10				0000} {0 ){						
(2) (1) (0) inst_mem_out_sig ra (31)	00000000 00000000 00000000 93810004 {00000000}{	00000000 00000000 00000000 238D000 1/17A (0000000) 1/17A		0006} { ){00	000000}		000000F}{10				D000} {0 }{						
(2) (1) (0) nst_mem_out_sig ra (31) (30)	00000000 0000000 0000000 93810004 {00000000} { 00000000}	00000000 00000000 00000000 238D0000 1 /17/4 (00000000) {10 00000000 10000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10				D000} {0 }						
(2) (1) (0) nst_mem_out_sig a (31) (30) (29)	00000000 00000000 00000000 93810004 {00000000}{ 10000000 10000000	00000000 00000000 00000000 23800000 \ 1174 (00000000) {10 00000000 10000000 10000000		0006} { ){00	0000000}		000000F} {10				D000} {0 }{						
(2) (1) (0) nst_mem_out_sig ra (31) (30) (29) (28)	0000000 0000000 0000000 93810004 {00000000} 10000000 10000000 0000000	00000000 00000000 2380000 1 117/ 00000000 1 1000000 10000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10				D000} {0 }{						
(2) (1) (0) nst_mem_out_sig ra (31) (30) (29) (23) (27)	0000000 0000000 0000000 93810004 {00000000} 10000000 10000000 0000000 0000000	00000000 00000000 238D0001 \1172 (00000000) 110 00000000 100000000 100000000 0000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F}{10				D000} {0 }{						
(2) (1) (0) nst_mem_out_sig ra (31) (29) (28) (27) (26)	0000000 0000000 0000000 93810004 {0000000} 10000000 10000000 0000000 0000000 000000	00000000 00000000 2380000 1174 (00000000 1174 (00000000 1100 10000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F}{10				0000} {0 }{						
(2) (1) (0) nst_mem_out_sig (31) (33) (28) (28) (27) (26) (25)	0000000 0000000 0000000 93810004 {00000000} 10000000 10000000 0000000 0000000 000000	00000000 00000000 00000000 23800000 1177 (00000000) 110 00000000 10000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10				⊃0000} {0 ){						
(2) (1) (0) nst_mem_out_sig ra (31) (23) (28) (27) (28) (27) (26) (25) (25) (25) (24)	0000000 0000000 0000000 93810004 (0000000) 10000000 10000000 0000000 0000000 000000	00000000 00000000 23800000 1 1722 (000000001 10 00000000 10 10000000 00000000 00000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10				0000} {0 )(						
(2) (1) (0) nst_mem_out_sig (31) (33) (23) (23) (24) (25) (24) (25) (24) (23)	00000000 00000000 93810004 {00000000} 10000000 10000000 0000000 0000000 000000	00000000 00000000 00000000 00000000 (000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	0000005} {10				0000} {0 }i						
(2) (1) (0) nst_mem_out_sig ra (31) (23) (26) (27) (25) (25) (24) (23) (24) (23)	00000000 0000000 0000000 93810004 (0000000) { 00000000 1000000 0000000 0000000 000000	00000000 00000000 23800000 23800000 10000000 10000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10				0000} {0 )(						
(2) (1) (0) nst_mem_out_sig ra (31) (23) (23) (25) (25) (24) (23) (23) (23) (23) (23) (23) (23) (23	00000000 00000000 00000000 93810004 (00000000) 10000000 10000000 00000000 00000000	00000000 00000000 23800000 10000000 10000000 10000000 1000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000=} {10				0000} {0 }(						
(2) (1) (0) nst_mem_out_sig a (31) (23) (23) (23) (24) (25) (24) (22) (22) (22) (22) (22) (22) (22	00000000 0000000 0000000 0000000 000000	00004000 00004000 00004000 23800001 1177 (00000000) 1177 (00000000) 110 00000000 00000000 00000000 0000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10				0000} {0 }(						
(2) (1) (0) nst_mem_out_sig a (31) (23) (23) (27) (25) (24) (24) (24) (24) (23) (22) (24) (24) (23) (22) (24) (21) (21) (21) (21) (21) (22) (22) (21) (22) (22	00000000 00000000 00000000 93810004 (00000000) 10000000 10000000 00000000 00000000	00000000 00000000 00000000 23800000 10000000 10000000 10000000 0000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} (10					0000000}.					
(2) (1) (0) nst_mem_out_sig a (31) (28) (27) (26) (27) (26) (27) (25) (24) (23) (22) (23) (22) (23) (22) (23) (22) (23) (23	00000000 00000000 93810004 (00000000) 10000000 00000000 00000000 00000000	00004000 00004000 00004000 23800000 100000000 10000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10						<u>}{ooooooo</u> ooo				
(2) (1) (0) nst_mem_out_sig a (31) (33) (23) (23) (25) (25) (25) (25) (24) (24) (23) (25) (25) (25) (25) (25) (25) (25) (25	00000000 00000000 00000000 93810004 (00000000) 10000000 10000000 00000000 00000000	00000000 00000000 00000000 22800000 10000000 10000000 10000000 1000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	00000007} {10					0000000}.				. ){000000	00} {10
(2) (0) (0) (0) (2) (3) (3) (3) (23) (23) (23) (24) (25) (24) (25) (24) (23) (23) (23) (23) (23) (23) (23) (23	00000000 00000000 93810004 (00000000) 10000000 10000000 00000000 00000000	00000000 00000000 00000000 23800000 100000000 10000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000=} (10					0000000}.	<u>}{ooooooo</u> ooo				00} {10
(2) (3) (0) nst_mem_out_sig a (31) (31) (23) (23) (25) (25) (24) (24) (23) (25) (24) (23) (24) (23) (23) (23) (23) (24) (23) (25)	00000000 00000000 00000000 00000000 0000	00000000 00000000 00000000 23800000 100000000 10000000 10000000 1000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000=} (10					0000000}.	<u>}{ooooooo</u> ooo			. ){000000	00} {10
(2) (0) (0) (0) (0) (0) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1	00000000 90000000 90000000 90000000 9000000	00000000 00000000 00000000 23800000 10000000 10000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10					0000000}.	<u>}{ooooooo</u> ooo			. ){000000	00} {10
(2) (1) (0) nst_mem_out_sig a (31) (30) (28) (27) (26) (28) (27) (26) (23) (25) (24) (23) (22) (23) (22) (23) (22) (23) (23	00000000 00000000 93810004 (00000000) 10000000 00000000 00000000 00000000	00006000 00006000 00006000 23800000 10000000 10000000 10000000 0000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	0000000*} (10					0000000}.	<u>}{ooooooo</u> ooo			. ){000000	00} {10
(2) (3) (3) (3) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2	00000000 00000000 00000000 00000000 000000	00000000 00000000 00000000 23800000 100000000 10000000 10000000 000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10					0000000}.	<u>}{ooooooo</u> ooo			. ){000000	00} {10
(2) (1) (0) nst_mem_out_sig (31) (33) (28) (28) (28) (28) (29) (29) (29) (29) (29) (29) (29) (29	00000000 00000000 93810004 (00000000) 10000000 10000000 00000000 00000000	00006000 00006000 00006000 23800000 100000000 100000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000F} {10					0000000}.	<u>}{ooooooo</u> ooo			. ){000000	00} {10
(2) (1) (0) nst_nem_out_sig (31) (31) (31) (33) (29) (27) (28) (27) (26) (29) (29) (24) (22) (21) (21) (21) (21) (14) (15) (14) (12) (12) (12) (13)	00000000 00000000 00000000 00000000 0000	000000000 000000000 00000000 238000000 100000000 100000000 100000000 1000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000=} {10					0000000}.	<u>}{ooooooo</u> ooo			. ){000000	00} {10
(2) (0) (0) (0) (1) (0) (2) (3) (3) (28) (28) (28) (28) (28) (28) (28) (28	00000000 00000000 93810004 (00000000) 10000000 10000000 00000000 00000000	00000000 00000000 00000000 23800000 100000000 10000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	00000001 (10					0000000}.	<u>}{ooooooo</u> ooo			. ){000000	
(2) (1) (0) nst_mem_out_sig (3) (3) (3) (28) (27) (28) (27) (28) (27) (28) (27) (28) (29) (29) (29) (21) (22) (21) (22) (22) (22) (23) (23) (24) (23) (23) (24) (23) (23) (24) (23) (23) (24) (23) (23) (24) (23) (23) (23) (23) (24) (23) (23) (23) (23) (23) (23) (23) (23	00000000 00000000 00000000 00000000 0000	000000000 000000000 00000000 238000000 100000000 100000000 100000000 1000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000=} {10					0000000}.	<u>}{ooooooo</u> ooo			. ){000000	
(2) (0) (0) (0) (1) (0) (2) (3) (3) (28) (28) (28) (28) (28) (28) (28) (28	00000000 93310004 (0000000) 93310004 (0000000) 10000000 00000000 00000000 00000000	00000000 00000000 00000000 23800000 100000000 10000000 10000000 00000000		0006} { ){00	0000000}	<u>}{0000000}</u> {1	000000=} (10					0000000}.	<u>}{ooooooo</u> ooo			. ){000000	00} {10

	0000000	0000000															
(6)	00000B0A	00000B0A															
(6) (5) (4) (3) (2)	03020100	03020100															
(4)	00000000	00000000															
(3)	00000000	00000000															
(2)	00000000	00000000															
(1)	00000000	00000000															
(0)	00000000	00000000								_						_	
(0)	00000000	0000000				Ţ											
_mem_out_sig	93B10004	0810)0212882	22	12	24.24000B	) <u>1140000</u>	0230	10020 Yo	8100018	A3D20000	22050001	23BD000	17AFf		130008	0383E823	3
_mem_out_ag		{00000000} {10															
(31)	00000000	00000000		0000771000		000000710	1000007 100	10000007 100	000007	10000000711	000000 7110.		100000017	100000077	100000	Moccocol	
(30)	10000000	1000000F														10000010	
(29)	10000000	10000007															<u>(</u> 10
(28)	00000000	00000000															100
(27)	00000000	00000000															
(26)		00000000															
(26) (25)	00000000	00000000															
(24)	00000000	00000000															
(23)		00000000															
23) 22)	00000000	00000000															
(21)	000000000	00000000															
(20)		00000000															
(19)	000000000	00000000						_			_						
(18)	00000000	0000000B						_		_	_	00000011					
17)		00000008						_			_	00000011					
(16)	00000000	00000003 0000000E															
10)	000000000								_						_		
15)		1000008															
14) 13)	00000000	0000000															
10)		0000000															
12)		0000000															
11) 10)	00000000	0000000															
	00000000	0000000								00000001							
9)	00000000	0000000															
3)	00000000	0000000															
7)	00000000	0000000															
6)	00000B0A	00000B0A															
	03020100	03020100															
	00000000	0000000															
	00000000	00000000															
2)	00000000	0000000															
	00000000	0000000															
	00000000																

inst_mem_out_sig	93B10004	8FA10000	8FA20004	8FA300	)8 (8FA4	000C (08	100022					
ra	{00000000} {1	<u>{00000000} {10</u>	000010} {10.		} <u>}</u> {000000	00} {{000	00000} \{0	0000000}	{00000000} .	){00000000	} {10000010}	{10000000} {0
-🔶 (31)	00000000	0000000										
	10000000	10000010										
-🔶 (29)	10000000	1000008			(100000	00						
-🔶 (28)	00000000	00000000										
-🔶 (27)	00000000	0000000										
-🔶 (26)	00000000	0000000										
-🔶 (25)	00000000	0000000										
-🔶 (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	0000000										
-🔷 (11)	00000000	0000000										
-🔷 (10)	00000000	00000001										
-🔷 (9)	00000000	0000000										
-🔶 (8)	00000000	0000000										
-🔷 (7)	00000000	0000000										
-🔶 (6)	00000B0A	00000B0A										
-🔷 (5)	03020100	03020100										
-🔶 (4)	00000000	00000000								(110A0908		
-🔶 (3)	00000000	0000000							0316FF10			
-🔶 (2)	0000000	0000000						0A0908				
-🔶 (1)	0000000	0000000				0302	0100					
-🔶 (0)	0000000	0000000										
 Now	652 71 us	minimi						humin	humun			