ESE 545 – Computer Architecture – Final Report

Spring 2016

Prepare by: Sanket Keni (ID No. 110451820)
And
Jay Nathani (ID No. 110422192)

Instruction Set Prepared

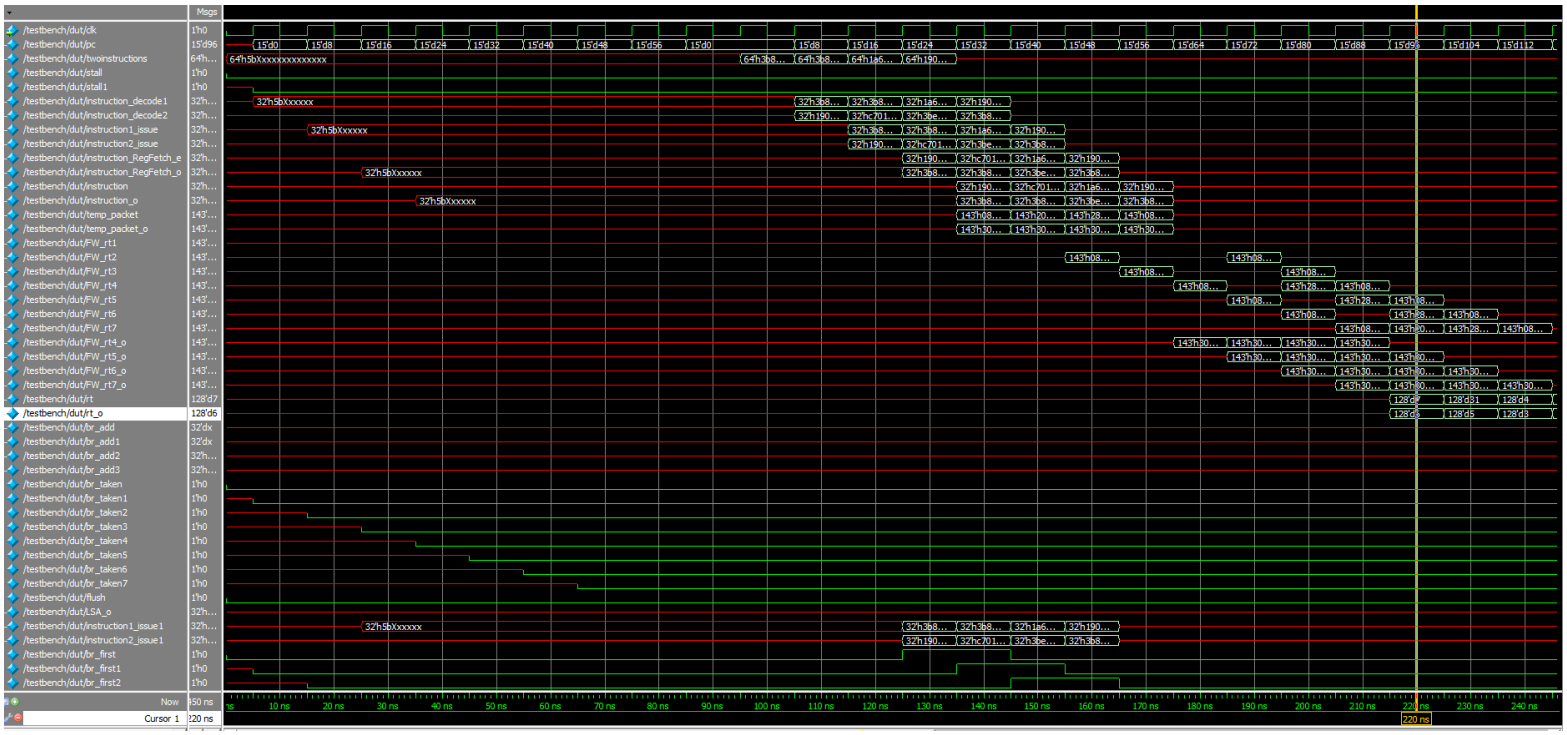
Sr. No.	Instruction	Syntax	Unit	Pipe	PipeDepth	Latency	UnitID
1	Load Quadword (d-form)	lqd rt,symbol(ra)	Local Store	Odd	6	6	7
2	Load Quadword (x-form)	lqx rt,ra,rb	Local Store	Odd	6	6	7
3	Load Quadword Instruction Relative (a-form)	lqr rt,symbol	Local Store	Odd	6	6	7
4	Store Quadword (d-form)	stqd rt,symbol(ra)	Local Store	Odd	6	6	7
5	Store Quadword (x-form)	stqx rt,ra,rb	Local Store	Odd	6	6	7
6	Store Quadword Instruction Relative (a-form)	stqrrt,symbol	Local Store	Odd	6	6	7
7	Immediate Load Halfword	ilh rt,symbol	Simple Fixed1	Even	2	2	1
8	Immediate Load Halfword Upper	ilhu rt,symbol	Simple Fixed1	Even	2	2	1
9	Immediate Load Word	il rt,symbol	Simple Fixed1	Even	2	2	1
10	Immediate Load Address	ila rt,symbol	Simple Fixed1	Even	2	2	1
11	Add Halfword	ah rt,ra,rb	Simple Fixed1	Even	2	2	1
12	Add Halfword Immediate	ahi rt,ra,value	Simple Fixed1	Even	2	2	1
13	Add Word	a rt,ra,rb	Simple Fixed1	Even	2	2	1
14	Add Word Immediate	ai rt,ra,value	Simple Fixed1	Even	2	2	1
15	Subtract from Halfword	sfh rt,ra,rb	Simple Fixed1	Even	2	2	1
16	Subtract from Halfword Immediate	sfhi rt,ra,value	Simple Fixed1	Even	2	2	1
17	Subtract from Word	sf rt,ra,rb	Simple Fixed1	Even	2	2	1
18	Subtract from Word Immediate	sfi rt,ra,value	Simple Fixed1	Even	2	2	1
19	Add Extended	addx rt,ra,rb	Simple Fixed1	Even	2	2	1
20	Subtract from Extended	sfx rt,ra,rb	Simple Fixed1	Even	2	2	1
21	Carry Generate	cg rt,ra,rb	Simple Fixed1	Even	2	2	1
22	Borrow Generate	bg rt,ra,rb	Simple Fixed1	Even	2	2	1
23	Multiply	mpy rt,ra,rb	Simple Fixed1	Even	2	2	1

24	Multiply Immediate	mpyi rt,ra,value	Simple Fixed1	Even	2	2	1
25	Multiply and Add	mpya rt,ra,rb,rc	Single Precision2	Even	7	7	4
26	Average Bytes	avgb rt,ra,rb	Byte	Even	3	4	5
27	Absolute Differences of Bytes	absdb rt,ra,rb	Byte	Even	3	4	5
28	Count Ones in Bytes	cntb rt,ra	Byte	Even	3	4	5
29	Sum Bytes into Halfwords	sumb rt,ra,rb	Byte	Even	3	4	5
30	And	and rt,ra,rb	Simple Fixed1	Even	2	2	1
31	And Byte Immediate	andbi rt,ra,value	Simple Fixed1	Even	2	2	1
32	Or	or rt,ra,rb	Simple Fixed1	Even	2	2	1
33	Or Byte Immediate	orbi rt,ra,value	Simple Fixed1	Even	2	2	1
34	Exclusive Or	xor rt,ra,rb	Simple Fixed1	Even	2	2	1
35	Exclusive Or Byte Immediate	xorbi rt,ra,value	Simple Fixed1	Even	2	2	1
36	Nand	nand rt,ra,rb	Simple Fixed1	Even	2	2	1
37	Nor	nor rt,ra,rb	Simple Fixed1	Even	2	2	1
38	Shift Left Halfword	shlh rt,ra,rb	Simple Fixed2	Even	3	4	2
39	Shift Left Halfword Immediate	shlhi rt,ra,value	Simple Fixed2	Even	3	4	2
40	Shift Left Word	shl rt,ra,rb	Simple Fixed2	Even	3	4	2
41	Shift Left Word Immediate	shli rt,ra,value	Simple Fixed2	Even	3	4	2
42	Shift Left Quadword by Bits	shlqbi rt,ra,rb	Permute	Odd	3	4	6
43	Shift Left Quadword by Bytes	shlqby rt,ra,rb	Permute	Odd	3	4	6
44	Rotate Quadword by Bytes	rotqby rt,ra,rb	Permute	Odd	3	4	6
45	Rotate Halfword	roth rt,ra,rb	Simple Fixed2	Even	3	4	2
46	Rotate Halfword Immediate	rothi rt,ra,value	Simple Fixed2	Even	3	4	2
47	Rotate Word	rot rt,ra,rb	Simple Fixed2	Even	3	4	2
48	Rotate Word Immediate	roti rt,ra,value	Simple Fixed2	Even	3	4	2
49	Halt If Equal	heq ra,rb	Branch	Odd	3	4	8
50	Halt If Equal Immediate	heqi ra,symbol	Branch	Odd	3	4	8
51	Halt if Greater than	hgt ra,rb	Branch	Odd	3	4	8

52	Halt if Greater than Immediate	hgti ra,symbol	Branch	Odd	3	4	8
53	Compare Equal Byte	ceqb rt,ra,rb	Simple Fixed1	Even	2	2	1
54	Compare Equal Byte Immediate	ceqbi rt,ra,value	Simple Fixed1	Even	2	2	1
55	Compare Greater Than Byte	cgtb rt,ra,rb	Simple Fixed1	Even	2	2	1
56	Compare Greater Than Byte Immediate	cgtbi rt,ra,value	Simple Fixed1	Even	2	2	1
57	Branch Relative	br symbol	Branch	Odd	3	4	8
58	Branch Absolute	bra symbol	Branch	Odd	3	4	8
59	Branch Absolute and Set Link	brasl rt,symbol	Branch	Odd	3	4	8
60	Branch If Not Zero Word	brnz rt,symbol	Branch	Odd	3	4	8
61	Branch If Zero Word	brz rt,symbol	Branch	Odd	3	4	8
62	Branch If Not Zero Halfword	brhnz rt,symbol	Branch	Odd	3	4	8
63	Branch If Zero Halfword	brhz rt,symbol	Branch	Odd	3	4	8
64	Floating Add	fa rt,ra,rb	Single Presion1	Even	6	6	3
65	Floating Subtract	fs rt,ra,rb	Single Presion1	Even	6	6	3
66	Floating Multiply	fm rt,ra,rb	Single Presion1	Even	6	6	3
67	Floating Multiply and Add	fma rt,ra,rb,rc	Single Presion1	Even	6	6	3
68	Floating Multiply and Subtract	fms rt,ra,rb,rc	Single Presion1	Even	6	6	3
69	Floating Compare Equal	fceq rt,ra,rb	Single Presion1	Even	6	6	3
70	Floating Compare Greater Than	fcgt rt,ra,rb	Single Presion1	Even	6	6	3
71	No Operation (Execute)	nop		Even			
72	No Operation (Load)	lnop		Odd			
73	Instruction Miss			Odd		6	

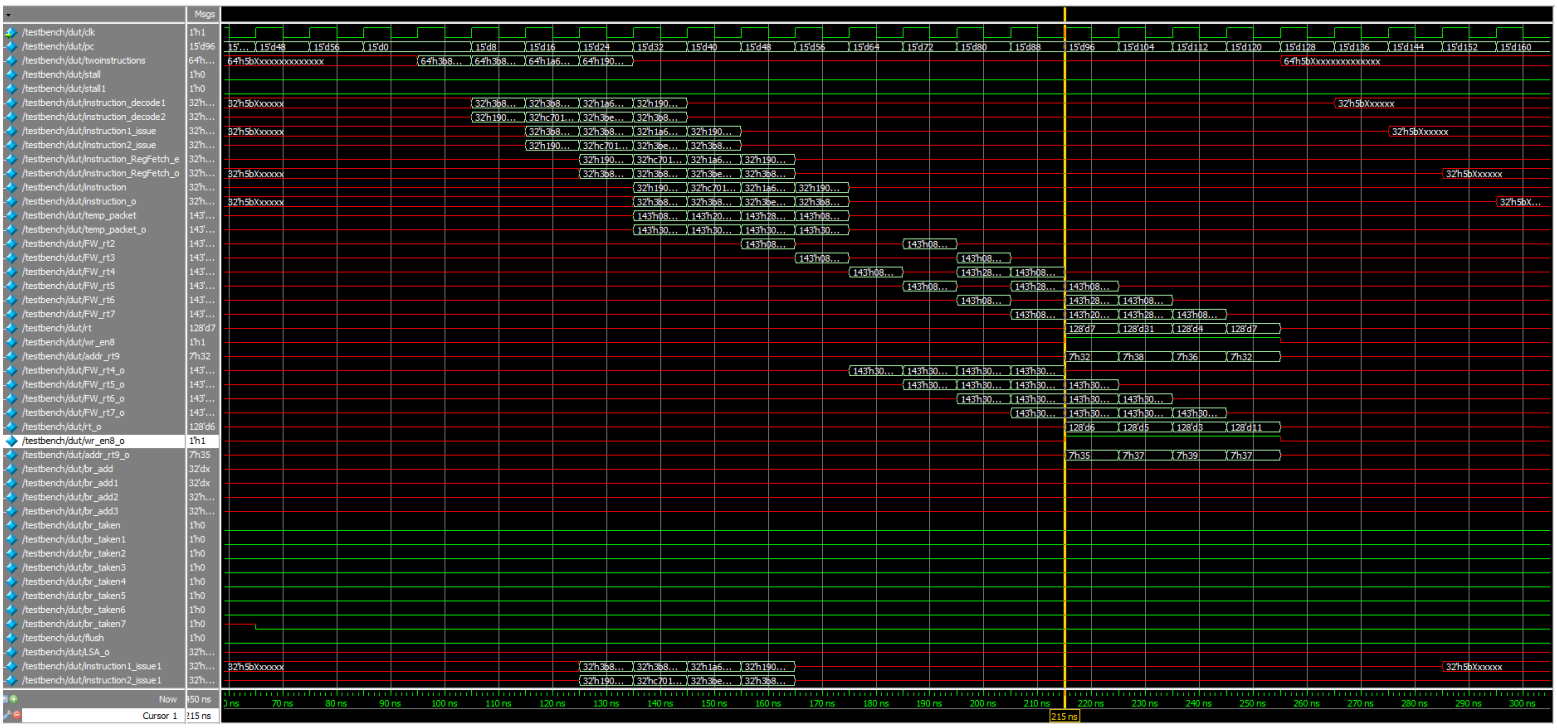
Loading of the memory with instructions and data ILB instruction load

The instructions and data are loaded in the main memory after the assembly code has been passed through the parser and the equivalent op-codes have been generated for each instruction. The output of the parser gives the exact 32 – bit instructions as needed in the SPU lite program. After the instructions are loaded in the memory, the cache would be empty, and so the first instruction to run will be *instruction miss*. *Instruction miss* takes a 128 byte block from the main memory and places it in the cache memory. This happens every time in the beginning of the program as the cache is empty, in the end of instructions in the cache and during a branch instruction when the tag value of the cache memory block does not match the PC (Program Counter) tag. The PC would be pointing at the branch target address generated during the execution of the branch instruction. An implementation of loading of the memory with instruction and data ILB instruction load can be seen in the following waveform:



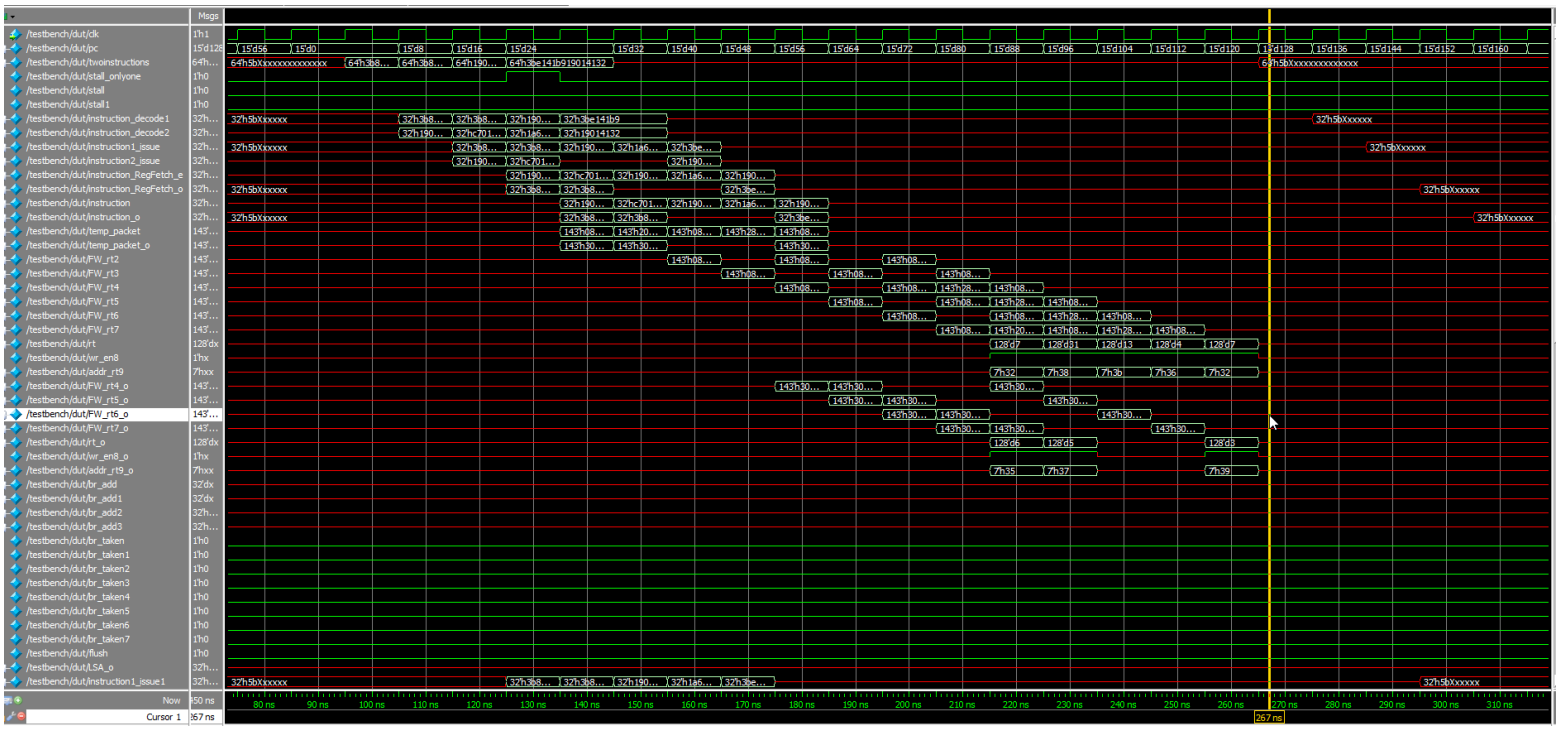
Dual – instruction fetch, decode, issue and execute (no hazards)

After the instructions and data are entered into the cache memory, as we are designing a two issue pipeline, in the dual-instruction fetch stage, two instructions per clock cycle will be fetched from the cache. These instructions are forwarded to the decode stage in the next clock cycle. The decode stage decodes the instructions into even and odd instructions and forwards it to the issue stage in the next clock cycle. The issue stage sends the instructions in their respective pipes (even instructions in even pipe and odd instructions in odd pipe) and forwards them to the register file. The register file gets the data of the registers to be used in the instruction and the result is computed and stored in the target register *rt*. Based on the type of instruction the latency will be different. After the beginning of execution, after eight cycles, the result is stored in *rt*, and is available for use. An implementation of dual – instruction fetch, decode, issue and execute (no – hazards) can be seen in the following waveform:



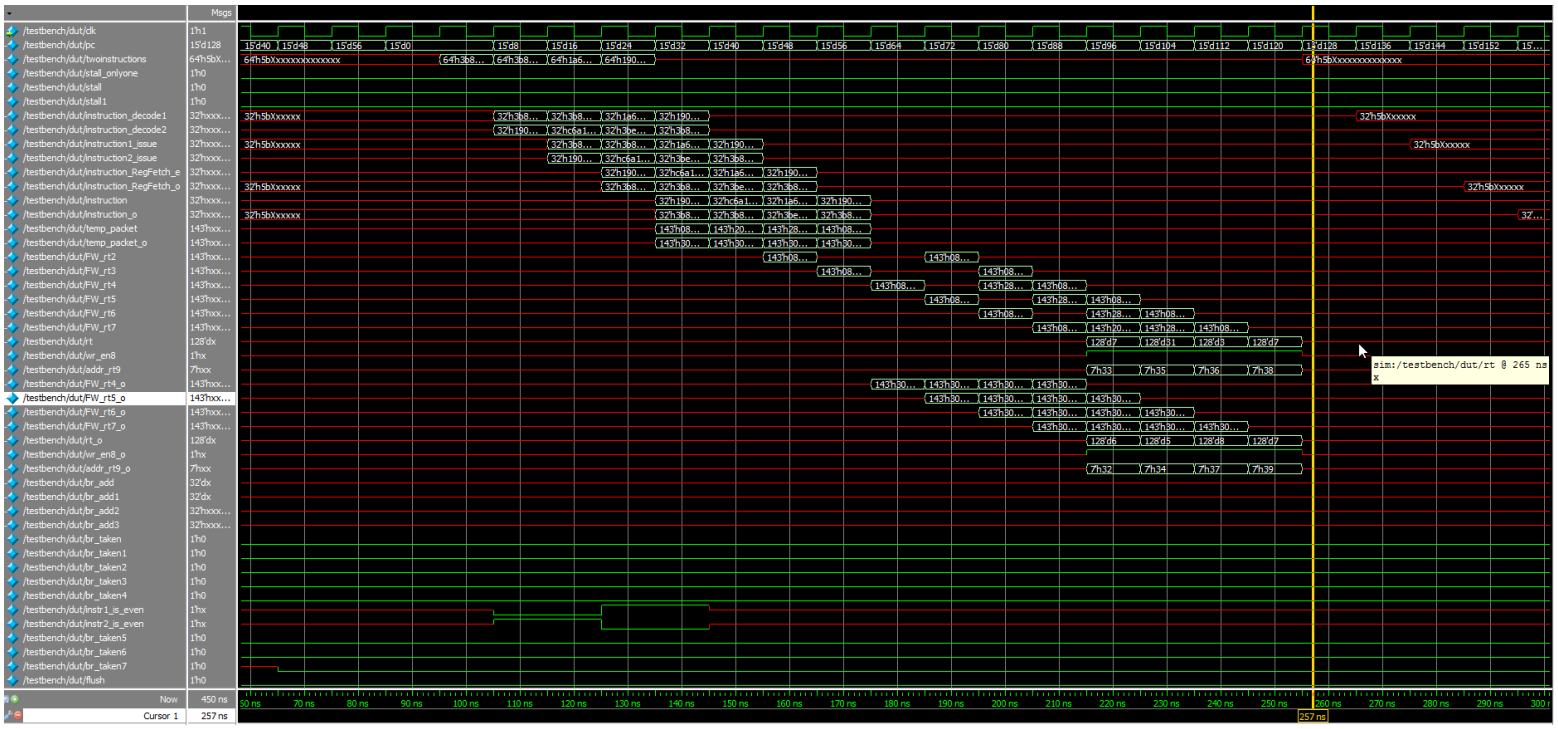
Structural hazard resolution

In the two issue pipeline structure, when two instructions fetched belong to the same pipe, a structural hazard occurs. This is because, at a time, only one instruction can access one pipe. This hazard is detected in the decode stage. In this case, a stall is generated. The fetch stage will not fetch any instructions in this case and the PC (Program Counter) is frozen. The first instruction is sent first in the respective pipe and the second instruction is stored. In the second cycle, the second instruction is sent in the pipe. The instruction no – op (no operation) is given in the other pipe until the fetch of the next instructions. For example, if the two instructions belong to the even pipe, the odd pipe will have no – op and vice versa. Without hazard the output of the two instructions should come together, but in this case the output of the second instruction is delayed by one clock cycle, one clock cycle is wasted in this case. An implementation of structural hazard and its resolution is depicted in the following waveform:

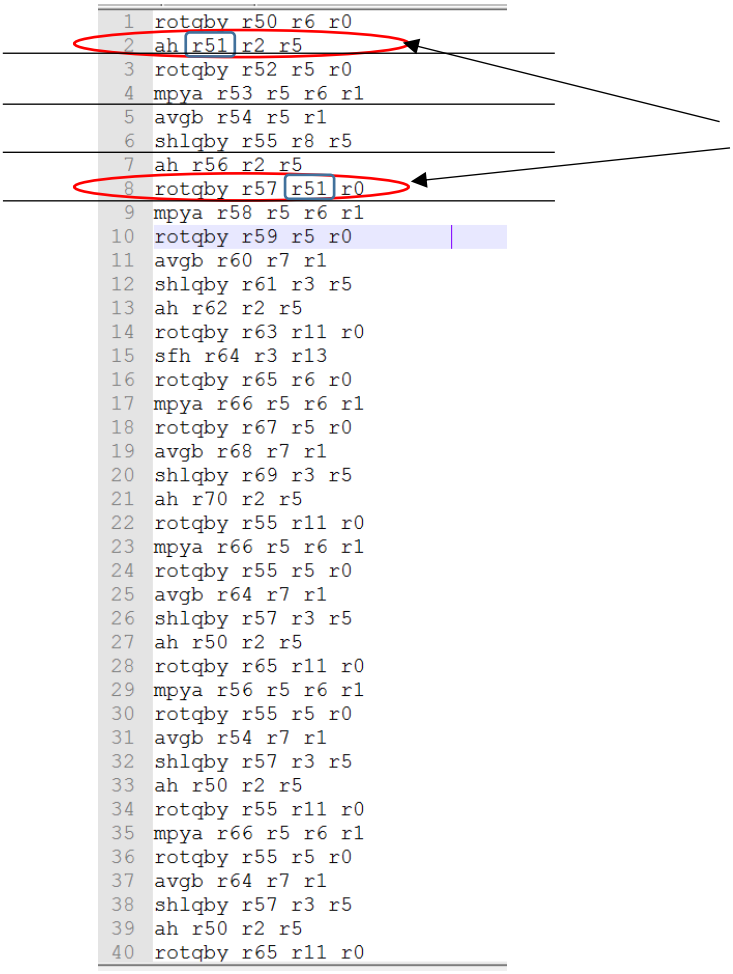


Data hazard resolution by forwarding (no stall)

Data hazards, also known as data dependent hazards, are the hazards in which target register of one instruction is used as the source register for a later fetched instruction. This type of hazard is also known as RAW (Read After Write) hazard. Broadly, data hazards can be removed by forwarding and in some cases by stall. If the instruction, that needs the output of a previously computed instruction, is fetched after some *n* cycles (*n* is the latency of the previous instruction), then this data hazard can be satisfied by forwarding the result of the previous instruction after those number of cycles. For example, if an ah (add half-word) instruction is fetched and its result is to be stored in r1, and there is another instruction sfh (subtract from half-word) which uses r1 as a source register and is fetched after three clock cycles, the computed value of the r1 is forwarded to sfh without any stall. This is called resolution by forwarding. An implementation of the data hazards and their resolution by forwarding without stalls is seen in the following waveform:



Following are a small part of instructions that were used to generate the above output:

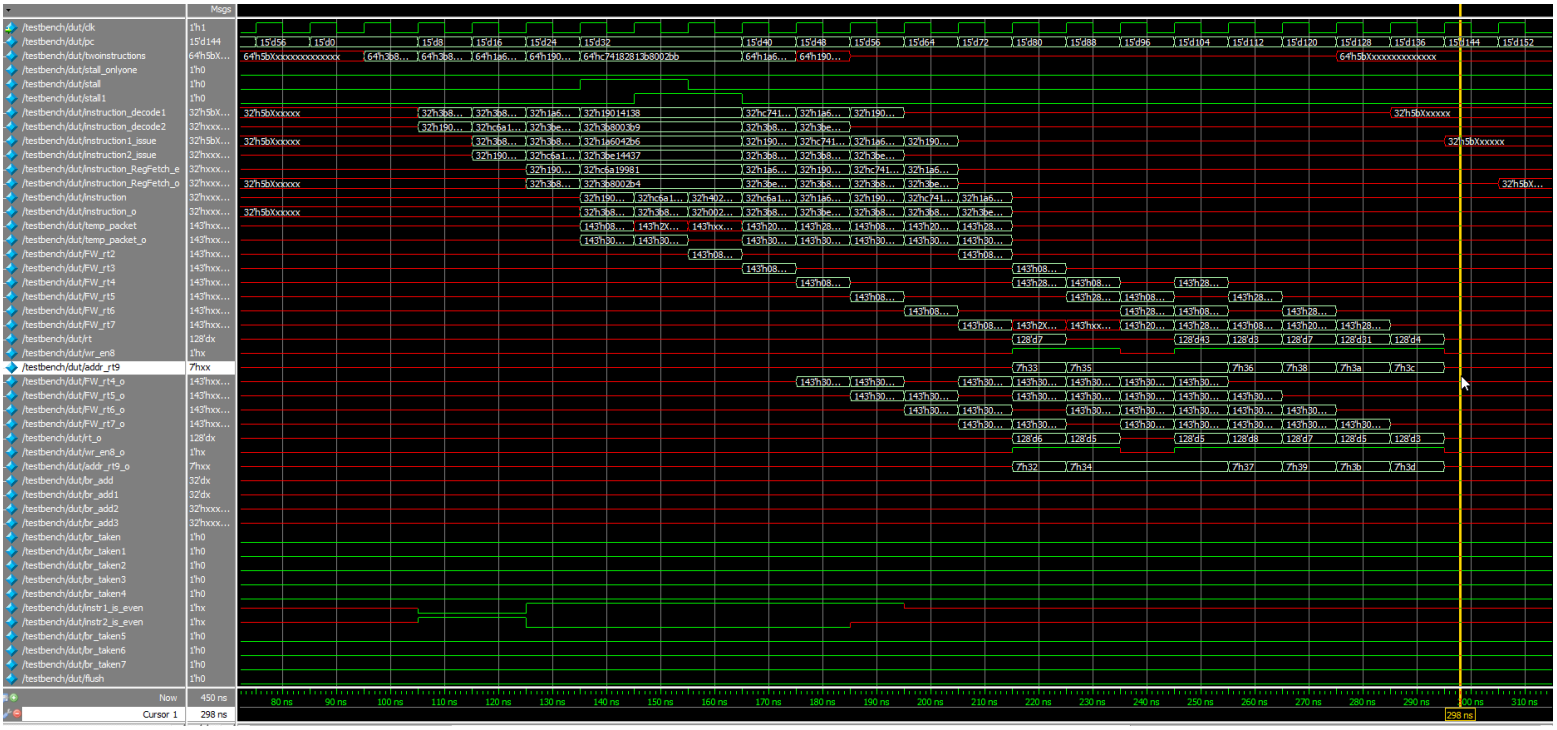


```
1  rotqby r50 r6 r0
2  ah r51 r2 r5
3  rotqby r52 r5 r0
4  mpya r53 r5 r6 r1
5  avgb r54 r5 r1
6  shlqby r55 r8 r5
7  ah r56 r2 r5
8  rotqby r57 r51 r0
9  mpya r58 r5 r6 r1
10 rotqby r59 r5 r0
11 avgb r60 r7 r1
12 shlqby r61 r3 r5
13 ah r62 r2 r5
14 rotqby r63 r11 r0
15 sfh r64 r3 r13
16 rotqby r65 r6 r0
17 mpya r66 r5 r6 r1
18 rotqby r67 r5 r0
19 avgb r68 r7 r1
20 shlqby r69 r3 r5
21 ah r70 r2 r5
22 rotqby r55 r11 r0
23 mpya r66 r5 r6 r1
24 rotqby r55 r5 r0
25 avgb r64 r7 r1
26 shlqby r57 r3 r5
27 ah r50 r2 r5
28 rotqby r65 r11 r0
29 mpya r56 r5 r6 r1
30 rotqby r55 r5 r0
31 avgb r54 r7 r1
32 shlqby r57 r3 r5
33 ah r50 r2 r5
34 rotqby r55 r11 r0
35 mpya r66 r5 r6 r1
36 rotqby r55 r5 r0
37 avgb r64 r7 r1
38 shlqby r57 r3 r5
39 ah r50 r2 r5
40 rotqby r65 r11 r0
```

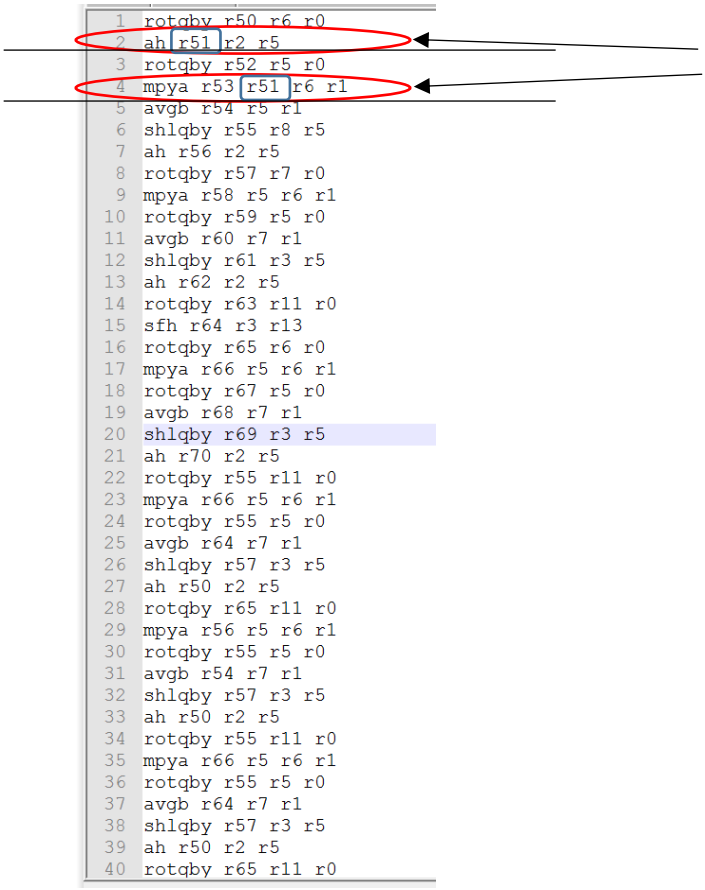
Here, data hazard is noticed in the instruction 2 between instruction 8. It is also noticed that r51 is next used after 2 bunches of instructions are fetched. This means that it occurs after two clock cycles and this hazard can be resolved by forwarding and no stall is needed.

Data hazard resolution by stalling and forwarding

As mentioned previously, the other method to resolve a data hazard is by stalling and forwarding. This method is used when the target register of one instruction is the source register of a later instruction. In this case the later instruction is fetched within *n* clock cycles (*n* is the latency of the previous instruction). In such a case, the program is stalled until the n^{th} clock cycle is achieved and after that the result is forwarded to the instruction that requires it. Hence the name, stalling and forwarding. An implementation of data hazards and its resolution with stalling and forwarding is evident in the following waveform:



Following are a small part of instructions that were used to generate the above output:

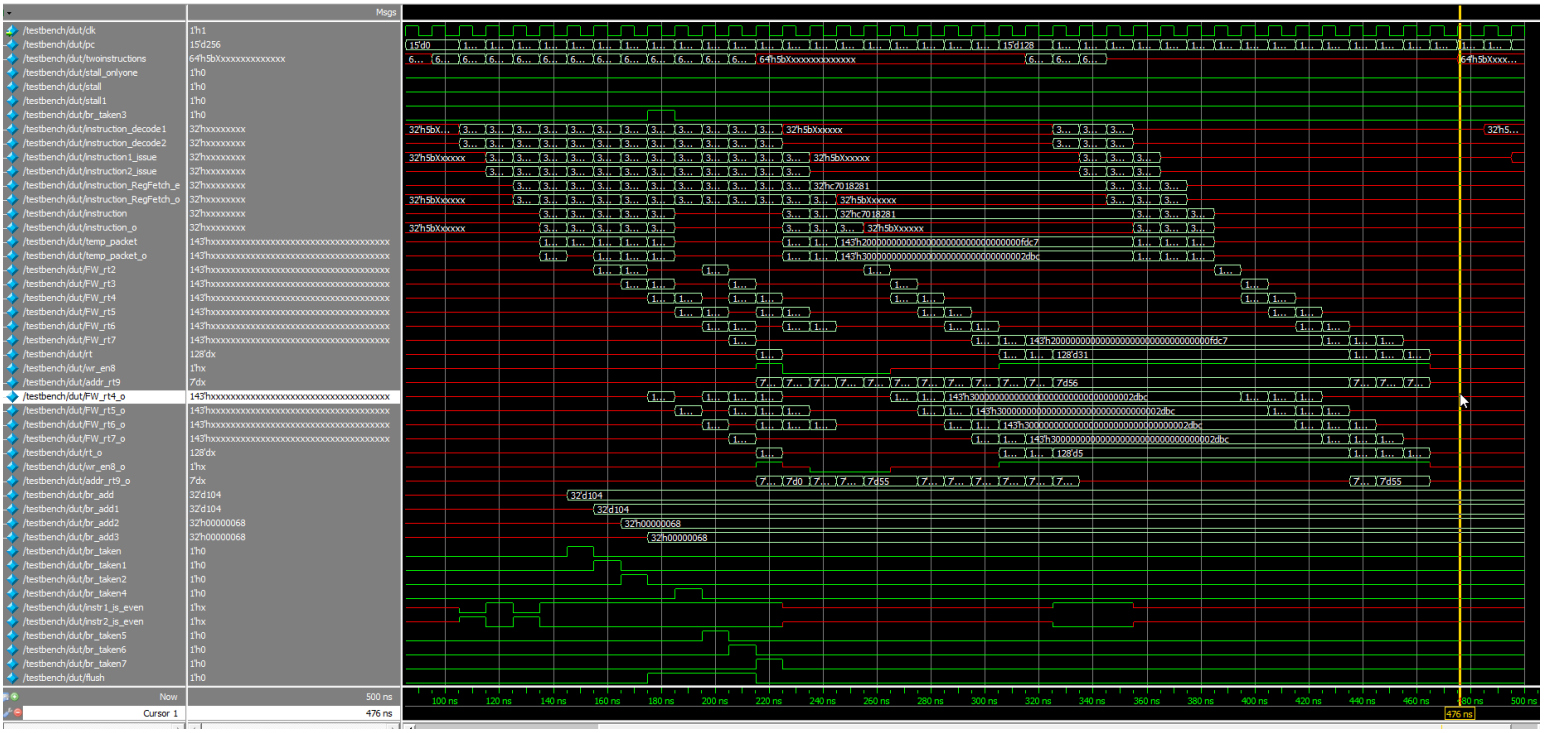


```
1  rotqby r50 r6 r0
2  ah r51 r2 r5
3  rotqby r52 r5 r0
4  mpya r53 r51 r6 r1
5  avgb r54 r5 r1
6  shlqby r55 r8 r5
7  ah r56 r2 r5
8  rotqby r57 r7 r0
9  mpya r58 r5 r6 r1
10 rotqby r59 r5 r0
11 avgb r60 r7 r1
12 shlqby r61 r3 r5
13 ah r62 r2 r5
14 rotqby r63 r11 r0
15 sfh r64 r3 r13
16 rotqby r65 r6 r0
17 mpya r66 r5 r6 r1
18 rotqby r67 r5 r0
19 avgb r68 r7 r1
20 shlqby r69 r3 r5
21 ah r70 r2 r5
22 rotqby r55 r11 r0
23 mpya r66 r5 r6 r1
24 rotqby r55 r5 r0
25 avgb r64 r7 r1
26 shlqby r57 r3 r5
27 ah r50 r2 r5
28 rotqby r65 r11 r0
29 mpya r56 r5 r6 r1
30 rotqby r55 r5 r0
31 avgb r54 r7 r1
32 shlqby r57 r3 r5
33 ah r50 r2 r5
34 rotqby r55 r11 r0
35 mpya r66 r5 r6 r1
36 rotqby r55 r5 r0
37 avgb r64 r7 r1
38 shlqby r57 r3 r5
39 ah r50 r2 r5
40 rotqby r65 r11 r0
```

Here, data hazard is noticed in the instruction 2 and instruction 4. It is also noticed that r51 is used next in the second bunch of fetched instructions. Since this bunch is not 2 clock cycles away (latency of ah), to resolve such a hazard, stalling and forwarding will be required.

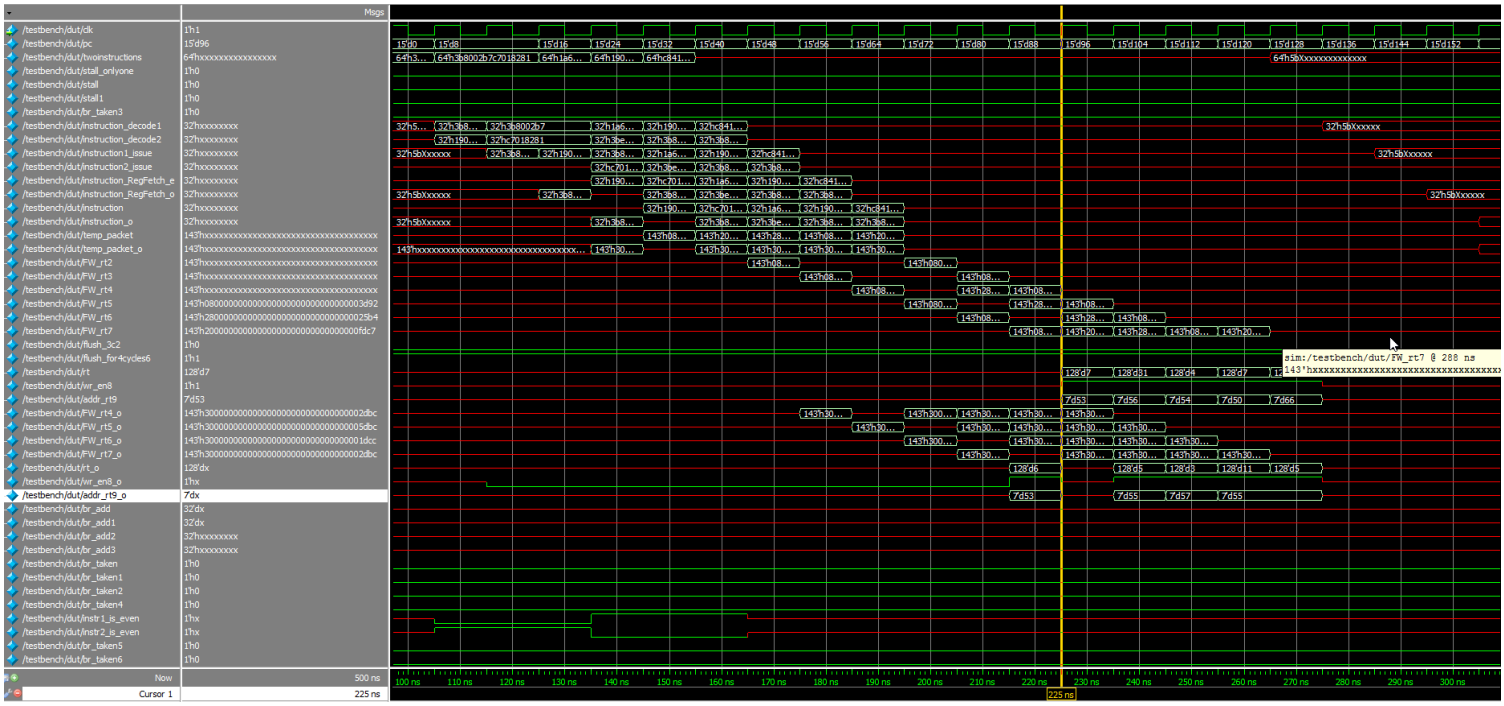
Control hazard resolution for branches

The control hazards for branches occur when the branch is predicted to be taken/not taken and ends up being not taken/taken. In either of the cases, there are some unwanted instructions that are executed before it is known if the branch is taken or not. The time taken is around four clock cycles from the starting of execution of the branch instruction. These instructions are needed to be flushed. The PC (Program Counter) points to the branch target address and does not wait for the flush to complete. An implementation of the control hazards for branches and its resolution using flush is seen in the following waveform:

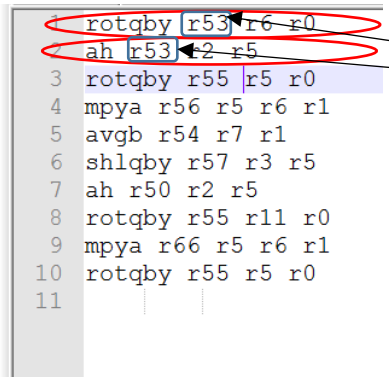


Other hazards – Write After Write Hazard

The Write After Write hazard occurs when an instruction *j*, that is executed after instruction *i*, tries to write the same destination register as the one written in instruction *i*. In the two issue architecture, when two instructions are fetched in one clock cycle, and both of them have the same target register, the WAW hazard is occurred. An implementation of the Write After Write hazard and its resolution using stalls is depicted in the following waveform:



Following are the instructions that were used to generate the above output:



These two instructions are fetched at the same time and written in the same register r53.