Stony Brook University

Cell SPU

ESE 545 Computer Architecture - Spring 2024

Date,

2/22/2024

Professor, Dorojevets

Noah Merone

Table of Contents

Acknowledgments:

Acknowledgment:

I would like to express my gratitude to Professor Dorojevets for his invaluable guidance, support, and expertise throughout the duration of this Computer Architecture course. Despite my limited availability for office hours, Professor Dorojevets's vast knowledge and dedication were important in navigating the complexities of computer architecture and enhancing my understanding of the subject.

Professor Dorojevets's background in physics, electronic engineering, and computer engineering, and computer architecture, highlighted by his achievements in both academic and industrial realms, served as an endless source of inspiration and motivation for me. His experience, particularly in the realms of parallel computer architecture and microprocessor design, have provided me with invaluable insight that helped my learning experience and influenced my approach to the course material.

Furthermore, I am grateful for Professor Dorojevets's commitment to having a strong learning environment and his willingness to go above and beyond to ensure that students have the necessary resources and support to succeed (lab computers, office hours, quick email replies). His passion for teaching and dedication to his students prove his spirit of academic excellence and mentorship. In addition, Professor Dorojevets shaped the curriculum of ESE 545 Computer

Architecture to provide me with invaluable knowledge and skills in quantitative analysis and evaluation of modern computer systems.

In conclusion, I am immensely pleased to have Professor Dorojevets for his expertise, support, and dedication to educational excellence. I am deeply appreciative of the opportunity to learn from such an educator and researcher. He will leave a lasting impression on my academic journey. Thank you, Professor Dorojevets!

Sincerely,

Noah Merone

SPU table:

No.	Name	Mnemonic	RTL Description	Exec Unit	Exec Pipe	Latency
			Simple Fixed 1:			
1	Add Halfword	ah rt,ra,rb	RT0:1 ← RA0:1 + RB0:1 RT2:3 ← RA2:3 + RB2:3 RT4:5 ← RA4:5 + RB4:5 RT6:7 ← RA6:7 + RB6:7 RT8:9 ← RA8:9 + RB8:9 RT10:11 ← RA10:11 + RB10:11 RT12:13 ← RA12:13 + RB12:13 RT14:15 ← RA14:15 + RB14:15	FX1	even	3
2	Add Halfword Immediate	ahi rt,ra,value	s ← RepLeftBit(I10,16) RT0:1 ← RA0:1 + s RT2:3 ← RA2:3 + s RT4:5 ← RA4:5 + s RT6:7 ← RA6:7 + s RT8:9 ← RA8:9 + s RT10:11 ← RA10:11 + s RT12:13 ← RA12:13+ s RT14:15 ← RA14:15 + s	FX1	even	3
3	Add Word	a rt,ra,rb	RT0:3 ← RA0:3 + RB0:3 RT4:7 ← RA4:7 + RB4:7 RT8:11 ← RA8:11 + RB8:11 RT12:15 ← RA12:15 + RB12:15	FX1	even	3
4	Add Word Immediate	ai rt,ra,value	t ← RepLeftBit(I10,32) RT0:3 ← RA0:3 + t RT4:7 ← RA4:7 + t RT8:11 ← RA8:11 + t RT12:15 ← RA12:15 + t	FX1	even	3
5	Subtract from Halfword	sfh rt,ra,rb	RT0:1 ← RB0:1 + (¬RA0:1) + 1 RT2:3 ← RB2:3 + (¬RA2:3) + 1 RT4:5 ← RB4:5 + (¬RA4:5) + 1 RT6:7 ← RB6:7 + (¬RA6:7) + 1 RT8:9 ← RB8:9 + (¬RA8:9) + 1 RT10:11 ← RB10:11 + (¬RA10:11) + 1 RT12:13 ← RB12:13 + (¬RA12:13) + 1 RT14:15 ← RB14:15 + (¬RA14:15) + 1	FX1	even	3

6	Subtract from Halfword Immediate	sfhi rt,ra,value	t ← RepLeftBit(I10,16) RT0:1 ← t + (¬RA0:1) + 1 RT2:3 ← t + (¬RA2:3) + 1 RT4:5 ← t + (¬RA4:5) + 1 RT6:7 ← t + (¬RA6:7) + 1 RT8:9 ← t + (¬RA8:9) + 1 RT10:11 ← t + (¬RA10:11) + 1 RT12:13 ← t + (¬RA12:13) + 1 RT14:15 ← t + (¬RA14:15) + 1	FX1	even	3
7	Subtract from Word	sf rt,ra,rb	RT0:3 ← RB0:3 + (¬RA0:3) + 1 RT4:7 ← RB4:7 + (¬RA4:7) + 1 RT8:11 ← RB8:11 + (¬RA8:11) + 1 RT12:15 ← RB12:15 + (¬RA12:15) + 1	FX1	even	3
8	Subtract from Word Immediate	sfi rt,ra,value	t ← RepLeftBit(I10,32) RT0:3 ← t + (¬RA0:3) + 1 RT4:7 ← t + (¬RA4:7) + 1 RT8:11 ← t + (¬RA8:11) + 1 RT12:15 ← t + (¬RA12:15) + 1	FX1	even	3
9	Add Extended	addx rt,ra,rb	RT0:3 ← RA0:3 + RB0:3 + RT31 RT4:7 ← RA4:7 + RB4:7 + RT63 RT8:11 ← RA8:11 + RB8:11 + RT95 RT12:15 ← RA12:15 + RB12:15 + RT127	FX1	even	3
10	Carry Generate	cg rt,ra,rb	for j = 0 to 15 by 4 t0:32 = ((0 || RAj::4) + (0 || RBj::4)) RTj::4 ← 310 || t0 end	FX1	even	3
11	Subtract from Extended	sfx rt,ra,rb	RT0:3 ← RB0:3 + (¬RA0:3) + RT31 RT4:7 ← RB4:7 + (¬RA4:7) + RT63 RT8:11 ← RB8:11 + (¬RA8:11) + RT95 RT12:15 ← RB12:15 + (¬RA12:15) + RT127	FX1	even	3
12	Borrow Generate	bg rt,ra,rb	for j = 0 to 15 by 4 if (RBj::4 ≥u RAj::4) then RTj::4 ← 1 else RTj::4 ← 0 end	FX1	even	3
13	Immediate Load Halfword	ilh rt,symbol	s ← I16 RT0:1 ← s RT2:3 ← s RT4:5 ← s RT6:7 ← s RT8:9 ← s RT10:11 ← s RT12:13 ← s RT14:15 ← s	FX1	even	3

14	Immediate Load Halfword Upper	ilhu rt,symbol	t ← I16 || 0x0000 RT0:3 ← t RT4:7 ← t RT8:11 ← t RT12:15 ← t	FX1	even	3
15	Immediate Load Word	il rt,symbol	t ← RepLeftBit(I16,32) RT0:3 ← t RT4:7 ← t RT8:11 ← t RT12:15 ← t	FX1	even	3
16	Immediate Load Address	ila rt,symbol	t ← 140 || I18 RT0:3 ← t RT4:7 ← t RT8:11 ← t RT12:15 ← t	FX1	even	3
17	Immediate Or Halfword Lower	iohl rt,symbol	t ← 0x0000 || I16 RT0:3 ← RT0:3 | t RT4:7 ← RT4:7 | t RT8:11 ← RT8:11 | t RT12:15 ← RT12:15 | t	FX1	even	3
18	Compare Equal Halfword	ceqh rt,ra,rb	for i = 0 to 15 by 2 If RAi::2 = RBi::2 then RTi::2 ← 0xFFFF else RTi::2 ← 0x0000 end	FX1	even	3
19	Compare Equal Halfword Immediate	ceqhi rt,ra,value	for i = 0 to 15 by 2 If RAi::2 = RepLeftBit(I10,16) then RTi::2 ← 0xFFFF else RTi::2 ← 0x0000 end	FX1	even	3
20	Compare Equal Word	ceq rt,ra,rb	for i = 0 to 15 by 4 If RAi::4 = RBi::4 then RTi::4 ← 0xFFFFFFFF else RTi::4 ← 0x00000000 end	FX1	even	3
21	Compare Equal Word Immediate	ceqi rt,ra,value	for i = 0 to 15 by 4 If RAi::4 = RepLeftBit(I10,32) then RTi::4 ← 0xFFFFFFFF else RTi::4 ← 0x00000000 end	FX1	even	3
22	Compare Greater Than Halfword	cgth rt,ra,rb	for i = 0 to 15 by 2 If RAi::2 > RBi::2 then RTi::2 ← 0xFFFF else RTi::2 ← 0x0000 end	FX1	even	3
23	Compare Greater Than Halfword Immediate	cgthi rt,ra,value	for i = 0 to 15 by 2 If RAi::2 > RepLeftBit(I10,16) then RTi::2 ← 0xFFFF else RTi::2 ← 0x0000	FX1	even	3

			end			
24	Compare Greater Than Word	cgt rt,ra,rb	for i = 0 to 15 by 4 If RAi::4 > RBi::4 then RTi::4 ← 0xFFFFFFFF else RTi::4 ← 0x00000000 end	FX1	even	3
25	Compare Greater Than Word Immediate	cgti rt,ra,value	for i = 0 to 15 by 4 If RAi::4 > RepLeftBit(I10,32) then RTi::4 ← 0xFFFFFFFF else RTi::4 ← 0x00000000 end	FX1	even	3
26	Immediate Load Halfword	ilh rt,symbol	s ← I16 RT0:1 ← s RT2:3 ← s RT4:5 ← s RT6:7 ← s RT8:9 ← s RT10:11 ← s RT12:13 ← s RT14:15 ← s	FX1	even	3
27	Immediate Load Halfword Upper	ilhu rt,symbol	t ← I16 || 0x0000 RT0:3 ← t RT4:7 ← t RT8:11 ← t RT12:15 ← t	FX1	even	3
28	Immediate Load Word	il rt,symbol	t ← RepLeftBit(I16,32) RT0:3 ← t RT4:7 ← t RT8:11 ← t RT12:15 ← t	FX1	even	3
29	Immediate Load Address	ila rt,symbol	t ← 140 || I18 RT0:3 ← t RT4:7 ← t RT8:11 ← t RT12:15 ← t	FX1	even	3
30	Compare Logical Greater Than Byte	clgtb rt,ra,rb	for i = 0 to 15 If RAi >u RBi thenRTi ← 0xFF else RTi ← 0x00 end	FX1	even	3
31	Compare Logical Greater Than Byte Immediate	clgtbi rt,ra,value	for i = 0 to 15 If RAi >u I102:9 then RTi ← 0xFF else RTi ← 0x00 end	FX1	even	3

32	Compare Logical Greater Than Halfword	clgth rt,ra,rb	for i = 0 to 15 by 2 If RAi::2 >u RBi::2 then RTi::2 ← 0xFFFF else RTi::2 ← 0x0000 end	FX1	even	3
33	Compare Logical Greater Than Halfword Immediate	clgthi rt,ra,value	for i = 0 to 15 by 2 If RAi::2 >u RepLeftBit(I10,16) then RTi::2 ← 0xFFFF else RTi::2 ← 0x0000 end	FX1	even	3
34	Compare Logical Greater Than Word	clgt rt,ra,rb	for i = 0 to 15 by 4 If RAi::4 >u RBi::4 then RTi::4 ← 0xFFFFFFFF else RTi::4 ← 0x00000000 end	FX1	even	3
35	Compare Logical Greater Than Word Immediate	clgti rt,ra,value	for i = 0 to 15 by 4 If RAi::4 >u RepLeftBit(I10,32) then RTi::4 ← 0xFFFFFFFF else RTi::4 ← 0x00000000 end	FX1	even	3
			Simple Fixed 2:			
36	Shift Left Halfword	shlh rt,ra,rb	for j = 0 to 15 by 2 s ← RBj::2 & 0x001F t ← RAj::2 for b = 0 to 15 if b + s < 16 then rb ← tb + s else rb ← 0 end RTj::2 ← r end	FX2	even	4
37	Shift Left Halfword Immediate	shlhi rt,ra,value	s ← RepLeftBit(I7,16) & 0x001F for j = 0 to 15 by 2 t ← RAj::2 for b = 0 to 15 if b + s < 16 then rb ← tb + s else rb ← 0 end RTj::2 ← r end	FX2	even	4

38	Shift Left Word	shl rt,ra,rb	for j = 0 to 15 by 4 s ← RBj::4 & 0x0000003F t ← RAj::4 for b = 0 to 31 if b + s < 32 then rb ← tb + s else rb ← 0 end RTj::4 ← r end	FX2	even	4
39	Shift Left Word Immediate	shli rt,ra,value	s ← RepLeftBit(I7,32) & 0x0000003F for j = 0 to 15 by 4 t ← RAj::4 for b = 0 to 31 if b + s < 32 then rb ← tb + s else rb ← 0 end RTj::4 ← r end	FX2	even	4
40	Rotate Word	rot rt,ra,rb	for j = 0 to 15 by 4 s ← RBj::4 & 0x0000001F t ← RAj::4 for b = 0 to 31 if b + s < 32 then rb ← tb + s else rb ← tb + s - 32 end RTj::4 ← r end	FX2	even	4
41	Rotate Word Immediate	roti rt,ra,value	s ← RepLeftBit(I7,32) & 0x0000001F for j = 0 to 15 by 4 t ← RAj::4 for b = 0 to 31 if b + s < 32 then rb ← tb + s else rb ← tb + s - 32 end RTj::4 ← r end	FX2	even	4
42	Rotate Halfword	roth rt,ra,rb	for j = 0 to 15 by 2 s ← RBj::2 & 0x000F t ← RAj::2 for b = 0 to 15 if b + s < 16 then rb ← tb + s else rb ← tb + s - 16 end RTj::2 ← r end	FX2	even	4

43	Rotate Halfword Immediate	rothi rt,ra,value	s ← RepLeftBit(I7,16) & 0x000F for j = 0 to 15 by 2 t ← RAj::2 for b = 0 to 15 if b + s < 16 then rb ← tb + s else rb ← tb + s - 16 end RTj::2 ← r end	FX2	even	4
			Single Precision			
44	Multiply	mpy rt,ra,rb	RT0:3 ← RA2:3 * RB2:3 RT4:7 ← RA6:7 * RB6:7 RT8:11 ← RA10:11 * RB10:11 RT12:15 ← RA14:15 * RB14:15	FP	even	8
45	Multiply Unsigned	mpyu rt,ra,rb	RT0:3 ← RA2:3 |*| RB2:3 RT4:7 ← RA6:7 |*| RB6:7 RT8:11 ← RA10:11 |*| RB10:11 RT12:15 ← RA14:15 |*| RB14:15	FP	even	8
46	Multiply Immediate	mpyi rt,ra,value	t ← RepLeftBit(I10,16) RT0:3 ← RA2:3 * t RT4:7 ← RA6:7 * t RT8:11 ← RA10:11 * t RT12:15 ← RA14:15 * t	FP	even	8
47	Multiply Unsigned Immediate	mpyui rt,ra,value	t ← RepLeftBit(I10,16) RT0:3 ← RA2:3 |*| t RT4:7 ← RA6:7 |*| t RT8:11 ← RA10:11 |*| t RT12:15 ← RA14:15 |*| t	FP	even	8
48	Multiply and Add	mpya rt,ra,rb,rc	t0 ← RA2:3 * RB2:3 t1 ← RA6:7 * RB6:7 t2 ← RA10:11 * RB10:11 t3 ← RA14:15 * RB14:15 RT0:3 ← t0 + RC0:3 RT4:7 ← t1 + RC4:7 RT8:11 ← t2 + RC8:11 RT12:15 ← t3 + RC12:15	FP	even	8

49	Multiply High	mpyh rt,ra,rb	t0 ← RA0:1 * RB2:3 t1 ← RA4:5 * RB6:7 t2 ← RA8:9 * RB10:11 t3 ← RA12:13 * RB14:15 RT0:3 ← t02:3 || 0x0000 RT4:7 ← t12:3 || 0x0000 RT8:11 ← t22:3 || 0x0000 RT12:15 ← t32:3 || 0x0000	FP	even	8
50	Multiply and Shift Right	mpys rt,ra,rb	t0 ← RA2:3 * RB2:3 t1 ← RA6:7 * RB6:7 t2 ← RA10:11 * RB10:11 t3 ← RA14:15 * RB14:15 RT0:3 ← RepLeftBit(t00:1,32) RT4:7 ← RepLeftBit(t10:1,32) RT8:11 ← RepLeftBit(t20:1,32) RT12:15 ← RepLeftBit(t30:1,32)	FP	even	8
51	Multiply High High	mpyhh rt,ra,rb	RT0:3 ← RA0:1 * RB0:1 RT4:7 ← RA4:5 * RB4:5 RT8:11 ← RA8:9 * RB8:9 RT12:15 ← RA12:13 * RB12:13	FP	even	8
52	Floating Add	fa rt,ra,rb		FP	even	8
53	Floating Subtract	fs rt,ra,rb		FP	even	8
54	Floating Multiply	fm rt,ra,rb		FP	even	8
55	Floating Multiply and Add	fma rt,ra,rb,rc		FP	even	8
56	Floating Negative Multiply and Subtract	fnms rt,ra,rb,rc		FP	even	8
57	Floating Multiply and Subtract	fms rt,ra,rb,rc		FP	even	8
Byte:						

58	Count Ones in Bytes	cntb rt,ra	for j = 0 to 15 c = 0 b ← RAj For m = 0 to 7 If bm = 1 then c ← c + 1 end RTj ← c end	Byte(4)	even	4
59	Average Bytes	avgb rt,ra,rb	for j = 0 to 15 RTj ← ((0x00 || RAj) + (0x00 || RBj) + 1)7:14 end	Byte(4)	even	4
60	Sum Bytes into Halfwords	sumb rt,ra,rb	RT0:1 ← RB0 + RB1+ RB2 + RB3 RT2:3 ← RA0 + RA1+ RA2 + RA3 RT4:5 ← RB4 + RB5+ RB6 + RB7 RT6:7 ← RA4 + RA5+ RA6 + RA7 RT8:9 ← RB8 + RB9+ RB10 + RB11 RT10:11 ← RA8 + RA9+ RA10 + RA11 RT12:13 ← RB12 + RB13+ RB14 + RB15 RT14:15 ← RA12 + RA13+ RA14 + RA15	Byte(4)	even	4
61	Absolute Differences of Bytes	absdb rt,ra,rb	for j = 0 to 15 if (RBj >u RAj) then RTj ← RBj - RAj else RTj ← RAj - RBj end	Byte(4)	even	4
			Permute:			
62	Shift Left Quadword by Bits	shlqbi rt,ra,rb	s ← RB29:31 for b = 0 to 127 if b + s < 128 then rb ← RAb + s else rb ← 0 end RT ← r	Permute(5)	odd	4
63	Shift Left Quadword by Bits Immediate	shlqbii rt,ra,value	s ← I7 & 0x07 for b = 0 to 127 if b + s < 128 then rb ← RAb + s else rb ← 0 end RT ← r	Permute(5)	odd	4

64	Shift Left Quadword by Bytes	shlqby rt,ra,rb	s ← RB27:31 for b = 0 to 15 if b + s < 16 then rb ← RAb + s else rb ← 0 end RT ← r	Permute(5)	odd	4
65	Shift Left Quadword by Bytes Immediate	shlqbyi rt,ra,value	s ← I7 & 0x1F for b = 0 to 15 if b + s < 16 then rb ← RAb + s else rb ← 0 end RT ← r	Permute(5)	odd	4
66	Shift Left Quadword by Bytes from Bit Shift Count	shlqbybi rt,ra,rb	s ← RB24:28 for b = 0 to 15 if b + s < 16 then rb ← RAb + s else rb ← 0x00 end RT ← r	Permute(5)	odd	4
67	Rotate Quadword by Bytes	rotqby rt,ra,rb	s ← RB28:31 for b = 0 to 15 if b + s < 16 then rb ← RAb + s else rb ← RAb + s - 16 end RT ← r	Permute(5)	odd	4
68	Rotate Quadword by Bytes Immediate	rotqbyi rt,ra,value	s ← I714:17 for b = 0 to 15 if b + s < 16 then rb ← RAb + s else rb ← RAb + s - 16 end RT ← r	Permute(5)	odd	4
69	Rotate Quadword by Bytes from Bit Shift Count	rotqbybi rt,ra,rb	s ← RB24:28 for b = 0 to 15 if b + s < 16 then rb ← RAb + s else rb ← RAb + s - 16 end RT ← r	Permute(5)	odd	4
	Local Store:					

70	Load Quadword (d-form)	lqd rt,symbol(ra)	LSA ← (RepLeftBit(I10 || 0b0000,32) + RA0:3) & LSLR & 0xFFFFFFF0 RT ← LocStor(LSA, 16)	LS	odd	1
71	Load Quadword (x-form)	lqx rt,ra,rb	LSA ← (RA0:3 + RB0:3) & LSLR & 0xFFFFFFF0 RT ← LocStor(LSA,16)	LS	odd	1
72	Load Quadword (a-form)	lqa rt,symbol	LSA ← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0 RT ← LocStor(LSA,16)	LS	odd	1
73	Load Quadword Instruction Relative (a-form)	lqr rt,symbol	LSA ← (RepLeftBit(I16 || 0b00,32) + PC) & LSLR & 0xFFFFFFF0 RT ← LocStor(LSA,16)	LS	odd	1
74	Store Quadword (d-form)	stqd rt,symbol(ra)	LSA ← (RepLeftBit(I10 || 0b0000,32) + RA0:3) & LSLR & 0xFFFFFFF0 LocStor(LSA,16) ← RT	LS	odd	1
75	Store Quadword (x-form)	stqx rt,ra,rb	LSA ← (RA0:3 + RB0:3) & LSLR & 0xFFFFFFF0 LocStor(LSA,16) ← RT	LS	odd	1
76	Store Quadword (a-form)	stqa rt,symbol	LSA ← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0 LocStor(LSA,16) ← RT	LS	odd	1
77	Store Quadword Instruction Relative (a-form)	stqr rt,symbol	LSA ← (RepLeftBit(I16 || 0b00,32) + PC) & LSLR & 0xFFFFFFF0 LocStor(LSA,16) ← RT	LS	odd	1
			Branch:			
78	Branch Relative	br symbol	PC ← (PC + RepLeftBit(I16 || 0b00,32)) & LSLR	Branch(7)	odd	1
79	Branch Absolute	bra symbol	PC ← RepLeftBit(I16 || 0b00,32) & LSLR	Branch(7)	odd	1
80	Branch Relative and Set Link	brsl rt,symbol	RT0:3 ← (PC + 4) & LSLR RT4:15 ← 0 PC ← (PC + RepLeftBit(I16 || 0b00,32)) & LSLR	Branch(7)	odd	1
81	Branch Absolute and Set Link	brasl rt,symbol	RT0:3 ← (PC + 4) & LSLR RT4:15 ← 0 PC ← RepLeftBit(I16 || 0b00,32) & LSLR	Branch(7)	odd	1
82	Branch Indirect	bi ra	PC ← RA0:3 & LSLR & 0xFFFFFFFC if (E = 0 and D = 0) then interrupt enable status is not modified else if (E = 1 and D = 0) then enable interrupts at target else if (E = 0 and D = 1) then disable interrupts at target else if (E = 1 and D = 1) then reserved	Branch(7)	odd	1

83	Branch If Not Zero Word	brnz rt,symbol	If RT0:3 ≠ 0 then
			PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
			else
			PC ← (PC+4) & LSLR
				Branch(7)	odd	1

84	Branch If Zero Word	brz rt,symbol	If RT0:3 = 0 then
			PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
			else
			PC ← (PC + 4) & LSLR
				Branch(7)	odd	1

85	Branch If Not Zero Halfword	brhnz rt,symbol	If RT2:3 ≠ 0 then
			PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
			else
			PC ← (PC + 4) & LSLR
				Branch(7)	odd	1

86	Branch If Zero Halfword	brhz rt,symbol	If RT2:3 = 0 then
			PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
			else
			PC ← (PC + 4) & LSLR
				Branch(7)	odd	1

87	Branch Indirect If Zero	biz rt,ra	t ← RA0:3 & LSLR & 0xFFFFFFFC
			u ← LSLR & (PC + 4)
			If RT0:3 = 0 then
			PC ← t & LSLR & 0xFFFF FFFC
			if (E = 0 and D = 0) then interrupt enable status is not modified
			else if (E = 1 and D = 0) then enable interrupts at target
			else if (E = 0 and D = 1) then disable interrupts at target
			else if (E = 1 and D = 1) then reserved
			else
			PC ← u
				Branch(7)	odd	1

88	Branch Indirect If Not Zero	binz rt,ra	t ← RA0:3 & LSLR & 0xFFFFFFFC
			u ← LSLR & (PC + 4)
			If RT0:3 != 0 then
			PC ← t & LSLR & 0xFFFFFFFC
			if (E = 0 and D = 0) then interrupt enable status is not modified
			else if (E = 1 and D = 0) then enable interrupts at target
			else if (E = 0 and D = 1) then disable interrupts at target
			else if (E = 1 and D = 1) then reserved
			else
			PC ← u
				Branch(7)	odd	1

89	Branch Indirect If Zero Halfword	bihz rt,ra	t ← RA0:3 & LSLR & 0xFFFFFFFC
			u ← LSLR & (PC + 4)
			If RT2:3 = 0 then do
			PC ← t & LSLR & 0xFFFFFFFC
			if (E = 0 and D = 0) then interrupt enable status is not modified
			else if (E = 1 and D = 0) then enable interrupts at target
			else if (E = 0 and D = 1) then disable interrupts at target
			else if (E = 1 and D = 1) then reserved
			else
			PC ← u
				Branch(7)	odd	1

90	Branch Indirect If Not Zero Halfword	bihnz rt,ra	t ← RA0:3 & LSLR & 0xFFFFFFFC
			u ← LSLR & (PC + 4)
			If RT2:3 != 0 then
			PC ← t & LSLR & 0xFFFFFFFC
			if (E = 0 and D = 0) then interrupt enable status is not modified
			else if (E = 1 and D = 0) then enable interrupts at target
			else if (E = 0 and D = 1) then disable interrupts at target
			else if (E = 1 and D = 1) then reserved
			else
			PC ← u
				Branch(7)	odd	1

			Control Instructions:			
91	No Operation (Load)	lnop		Operation	even	
92	No Operation (Execute)	nop		Operation	odd	
93	Stop and Signal	stop	PC ← PC + 4 & LSLR precise stop	Operation	odd	

Two processing pipes:

Odd Pipe:

The “Odd Pipe” module represents an important stage in a processor's pipeline, it is responsible for handling instructions and forwarding logic. In this module, it's the intermediary between instruction fetch (IF) and execution stages. This helps facilitate the execution of instructions and managing the data flow.

The module takes various inputs like clock signals, reset signals, operation codes, instruction formats, source and destination register addresses, immediate values, and even flags indicating register writes. It receives the data from preceding pipeline stages, like the program counter from the instruction fetch stage.

The module performs multiplexing to route instructions to different execution units based on the ‘unit’ input. It supports three execution units including Permute, Local Store, and Branch. Depending on the ‘unit’ input, the corresponding operation code, instruction format, and register write flag are set for the execution unit.

Additionally, the module manages forwarding paths to help data integrity and efficiency in the pipeline. It forwards the data and control signals between the pipeline stages, which help with timely execution of dependent instruction. Forwarding paths are updated based on the write back signals from the execution units, ensuring that the most recent data is ready for subsequent stages.

Overall, the “Odd Pipe” module plays an important role in the processor's pipeline by orchestrating instruction execution, managing data flow, and even optimizing the performance through efficient forwarding code.

Odd Pipe Code:

```
/*************************************************************************************************************
 * Module: Odd Pipe
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
```

* This module represents the odd pipeline stage in a processor, handling instructions and forwarding

* logic.

*---

* Inputs:

* - clock: Clock signal

* - reset: Reset signal

* - op_code: Operation code for the instruction

* - instr_format: Instruction format

* - unit: Destination unit for the instruction

* - dest_reg_addr: Destination register address

* - src_reg_a: Value of source register A

* - src_reg_b: Value of source register B

* - store_reg: Value of store register

* - imm_value: Immediate value

* - enable_reg_write: Flag indicating whether register write is enabled

* - program_counter_input: Program counter from IF stage

* - initial_: Flag indicating if the instruction is the initial_ in a pair

*---

* Outputs:

* - wb_data: Data to be written back

* - wb_reg_addr: Address of the register to be written back

* - wb_enable_reg_write: Flag indicating whether register write is enabled for write-back stage

* - program_counter_wb: New program counter for branch

* - branch_is_taken: Flag indicating whether branch is taken

* - disable_branch: If branch is taken and branch instruction is initial_ in pair, kill twin instruction

* - forwarded_data_wb: Forwarded data for write-back stage

* - forwarded_address_wb: Forwarded address for write-back stage

* - forwarded_write_flag_wb: Forwarded write flag for write-back stage

* - delayed_ls1_register_address: Delayed register addresses for ls1 unit

* - delayed_ls1_register_write: Delayed enable register write for ls1 unit

* - delayed_p1_register_address: Delayed register addresses for p1 unit

* - delayed_p1_register_write: Delayed enable register write for p1 unit

*---

* Internal Signals:

* - p1_op_code, ls1_op_code, br1_op_code: Operation codes for units

```
 *   - p1_instruction_format, ls1_instruction_format, br1_instruction_format: Instruction formats for units
 *   - p1_enable_register_write, ls1_enable_register_write, br1_enable_register_write:
 *     Enable register write flags for units
 *   - p1_output_stage4, ls1_output_stage6, br1_output_stage1: Output data for units
 *   - p1_output_register, ls1_output_register, br1_output_register: Output register addresses for units
 *   - p1_write_back, ls1_write_back, br1_write_back: Write flags for units
 ***************************************************************************************************/

module Odd_Pipe (
    clock,
    reset,
    op_code,
    instr_format,
    unit,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    store_reg,
    imm_value,
    enable_reg_write,
    program_counter_input,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    program_counter_wb,
    branch_is_taken,
    forwarded_data_wb,
    forwarded_address_wb,
    forwarded_write_flag_wb,
    initial_,
    disable_branch,
    delayed_ls1_register_address,
    delayed_ls1_register_write,
    delayed_p1_register_address,
    delayed_p1_register_write
```

```
);

  input clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  input [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  input [2:0] instr_format;
  // Execution unit of instruction
  input [1:0] unit;
  // Destination register address
  input [0:6] dest_reg_addr;
  // Values of source registers
  input [0:127] src_reg_a, src_reg_b, store_reg;
  // Immediate value, truncated based on instruction format
  input [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  input enable_reg_write;
  // Program counter from IF stage
  input [7:0] program_counter_input;
  // 1 if initial_ instruction in pair; used for determining order of branch
  input initial_;

  // Write Back Stage
  // Output value of Stage 7
  output logic [0:127] wb_data;
  // Destination register for write back data
  output logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic wb_enable_reg_write;
  // New program counter for branch
  output logic [7:0] program_counter_wb;
  // Was branch taken?
  output logic branch_is_taken;
```

```
// If branch is taken and branch instruction is initial_ in pair, kill twin instruction
output logic disable_branch;

// Internal Signals
// Staging register for forwarded values
output logic [6:0][0:127] forwarded_data_wb;
// Destination register for wb_data
output logic [6:0][0:6] forwarded_address_wb;
// Will wb_data write to RegTable
output logic [6:0] forwarded_write_flag_wb;
// Multiplexed opcode for p1
logic [0:10] p1_op_code;
// Multiplexed instr_format for p1
logic [2:0] p1_instruction_format;
// Multiplexed enable_reg_write for p1
logic p1_enable_register_write;
// Output value of p1 Stage 4
logic [0:127] p1_output_stage4;
// Destination register for wb_data from p1
logic [0:6] p1_output_register;
// Will wb_data from p1 write to RegTable
logic p1_write_back;
// Multiplexed opcode for ls1
logic [0:10] ls1_op_code;
// Multiplexed instr_format for ls1
logic [2:0] ls1_instruction_format;
// Multiplexed enable_reg_write for ls1
logic ls1_enable_register_write;
// Output value of ls Stage 6
logic [0:127] ls1_output_stage6;
// Destination register for wb_data from ls1
logic [0:6] ls1_output_register;
// Will wb_data from ls1 write to RegTable
logic ls1_write_back;
// Multiplexed opcode for br1
```

```
logic [0:10] br1_op_code;
// Multiplexed instr_format for br1
logic [2:0] br1_instruction_format;
// Multiplexed enable_reg_write for br1
logic br1_enable_register_write;
// Output value of br1 Stage 1
logic [0:127] br1_output_stage1;
// Destination register for wb_data from br1
logic [0:6] br1_output_register;
// Will wb_data from br1 write to RegTable
logic br1_write_back;

// Destination register for wb_data, delayed for ls1
output logic [5:0][0:6] delayed_ls1_register_address;
// Will wb_data write to RegTable, delayed for ls1
output logic [5:0] delayed_ls1_register_write;
// Destination register for wb_data, delayed for p1
output logic [3:0][0:6] delayed_p1_register_address;
// Will wb_data write to RegTable, delayed for p1
output logic [3:0] delayed_p1_register_write;

Permute p1 (
    .clock(clock),
    .reset(reset),
    .op_code(p1_op_code),
    .instr_format(p1_instruction_format),
    .dest_reg_addr(dest_reg_addr),
    .src_reg_a(src_reg_a),
    .src_reg_b(src_reg_b),
    .imm_value(imm_value),
    .enable_reg_write(p1_enable_register_write),
    .wb_data(p1_output_stage4),
    .wb_reg_addr(p1_output_register),
    .wb_enable_reg_write(p1_write_back),
    .branch_is_taken(branch_is_taken),
```

```
    .delayed_rt_addr(delayed_p1_register_address),
    .delayed_enable_reg_write(delayed_p1_register_write)
);
Local_Store ls1 (
    .clock(clock),
    .reset(reset),
    .op_code(ls1_op_code),
    .instr_format(ls1_instruction_format),
    .dest_reg_addr(dest_reg_addr),
    .src_reg_a(src_reg_a),
    .src_reg_b(src_reg_b),
    .store_reg(store_reg),
    .imm_value(imm_value),
    .enable_reg_write(ls1_enable_register_write),
    .wb_data(ls1_output_stage6),
    .wb_reg_addr(ls1_output_register),
    .wb_enable_reg_write(ls1_write_back),
    .branch_is_taken(branch_is_taken),
    .delayed_rt_addr(delayed_ls1_register_address),
    .delayed_enable_reg_write(delayed_ls1_register_write)
);
Branch br1 (
    .clock(clock),
    .reset(reset),
    .op_code(br1_op_code),
    .instr_format(br1_instruction_format),
    .dest_reg_addr(dest_reg_addr),
    .src_reg_a(src_reg_a),
    .src_reg_b(src_reg_b),
    .store_reg(store_reg),
    .imm_value(imm_value),
    .enable_reg_write(br1_enable_register_write),
    .program_counter_input(program_counter_input),
    .wb_data(br1_output_stage1),
    .wb_reg_addr(br1_output_register),
```

```
    .wb_enable_reg_write(br1_write_back),
    .program_counter_wb(program_counter_wb),
    .branch_is_taken(branch_is_taken),
    .initial_(initial_),
    .disable_branch(disable_branch)
);

always_comb begin
  p1_op_code = 0;
  p1_instruction_format = 0;
  p1_enable_register_write = 0;
  ls1_op_code = 0;
  ls1_instruction_format = 0;
  ls1_enable_register_write = 0;
  br1_op_code = 0;
  br1_instruction_format = 0;
  br1_enable_register_write = 0;

  // Multiplexer to determine which execution unit will process the instruction, based on the 'unit' input
  case (unit)
    // Case when the instruction is going to the Permute Unit (p1)
    2'b00: begin
      p1_op_code = op_code;
      p1_instruction_format = instr_format;
      p1_enable_register_write = enable_reg_write;
    end
    // Case when the instruction is going to the Local Store Unit 1 (ls1)
    2'b01: begin
      ls1_op_code = op_code;
      ls1_instruction_format = instr_format;
      ls1_enable_register_write = enable_reg_write;
    end
    // Case when the instruction is going to the Branch Unit 1 (br1)
    2'b10: begin
      br1_op_code = op_code;
```

```
      br1_instruction_format = instr_format;
      br1_enable_register_write = enable_reg_write;
    end
    // Default case: when the instruction is going to the Permute Unit (p1)
    default begin
      p1_op_code = op_code;
      p1_instruction_format = instr_format;
      p1_enable_register_write = enable_reg_write;
    end
  endcase
end

always_ff @(posedge clock) begin
  if (reset == 1) begin
    // Reset values
    wb_data <= 0;
    wb_reg_addr <= 0;
    wb_enable_reg_write <= 0;
    foreach (forwarded_data_wb[i]) forwarded_data_wb[i] <= 0;
    foreach (forwarded_address_wb[i]) forwarded_address_wb[i] <= 0;
    foreach (forwarded_write_flag_wb[i]) forwarded_write_flag_wb[i] <= 0;
  end else begin
    // Update values based on forwarding paths
    wb_data <= forwarded_data_wb[6];
    wb_reg_addr <= forwarded_address_wb[6];
    wb_enable_reg_write <= forwarded_write_flag_wb[6];

    // Forwarding paths
    for (int i = 6; i > 0; i = i - 1) begin
      forwarded_data_wb[i] <= forwarded_data_wb[i-1];
      forwarded_address_wb[i] <= forwarded_address_wb[i-1];
      forwarded_write_flag_wb[i] <= forwarded_write_flag_wb[i-1];
    end
    // Specific forwarding path replacements
    casez ({
```

```
      ls1_write_back, p1_write_back, br1_write_back
    })
      3'b100: begin
        forwarded_data_wb[5] <= ls1_output_stage6;
        forwarded_address_wb[5] <= ls1_output_register;
        forwarded_write_flag_wb[5] <= ls1_write_back;
      end
      3'b010: begin
        forwarded_data_wb[3] <= p1_output_stage4;
        forwarded_address_wb[3] <= p1_output_register;
        forwarded_write_flag_wb[3] <= p1_write_back;
      end
      3'b001: begin
        forwarded_data_wb[0] <= br1_output_stage1;
        forwarded_address_wb[0] <= br1_output_register;
        forwarded_write_flag_wb[0] <= br1_write_back;
      end
      default: begin
        forwarded_data_wb[0] <= 0;
        forwarded_address_wb[0] <= 0;
        forwarded_write_flag_wb[0] <= 0;
      end
    endcase
  end
 end
endmodule
```

Odd Pipe Test Bench:

The ‘Odd Pipe’ testbench module is designed to simulate the behavior of the odd pipeline stage in a processor. The purpose of this module is to generate input simulation and capture output responses to verify the correctness and functionality of the odd pipeline stage.

This module takes various inputs like clock signal, reset signal, operation code, instruction format, destination unit, source register values, immediate value, and flags indicating register write enablement and branch conditions. It also provides the outputs including data to be written back, register addresses for write back stage, program counter for branch, and flags indicating the status of a branch and forwarding data.

Additionally, this module instantiates the ‘Odd Pipe’ module, representing the odd pipeline stage, and connects its inputs and outputs to the corresponding signals of the testbench. It then generates clock signals and sets up the inputs for various test cases to simulate different operations such as shift, rotate, load, and store. The simulation progresses by toggling the clock and updating the inputs to the specific test cases.

Overall, this testbench facilitates testing and validation of the odd pipeline stage by providing a controlled environment to observe its behavior under different scenarios and input conditions. This helps to ensure the correct functioning of the processor's odd pipeline stage before integration into the complete processor design.

Odd Pipe Test Bench Code:

```
/*************************************************************************************************************
 * Module: Odd Pipe Testbench
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *    This testbench module simulates the behavior of the odd pipeline stage in a processor, providing input
 *    stimuli and capturing output responses.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
```

*   - clock: Clock signal

*   - reset: Reset signal

*   - op_code: Operation code for the instruction

*   - instr_format: Instruction format

*   - unit: Destination unit for the instruction

*   - dest_reg_addr: Destination register address

*   - src_reg_a: Value of source register A

*   - src_reg_b: Value of source register B

*   - store_reg: Value of store register

*   - imm_value: Immediate value

*   - enable_reg_write: Flag indicating whether register write is enabled

*   - program_counter_input: Program counter from IF stage

*   - initial_: Flag indicating if the instruction is the initial_ in a pair

*----------

* Outputs:

*   - wb_data: Data to be written back

*   - wb_reg_addr: Address of the register to be written back

*   - wb_enable_reg_write: Flag indicating whether register write is enabled for write-back stage

*   - program_counter_wb: New program counter for branch

*   - branch_is_taken: Flag indicating whether branch is taken

*   - disable_branch: If branch is taken and branch instruction is initial_ in pair, kill twin instruction

*   - forwarded_data_wb: Forwarded data for write-back stage

*   - forwarded_address_wb: Forwarded address for write-back stage

*   - forwarded_write_flag_wb: Forwarded write flag for write-back stage

*   - delayed_ls1_register_address: Delayed register addresses for ls1 unit

*   - delayed_ls1_register_write: Delayed enable register write for ls1 unit

*   - delayed_p1_register_address: Delayed register addresses for p1 unit

*   - delayed_p1_register_write: Delayed enable register write for p1 unit

*----------

* Internal Signals:

*   - p1_op_code, ls1_op_code, br1_op_code: Operation codes for units

*   - p1_instr_format, ls1_instr_format, br1_instr_format: Instruction formats for units

*   - p1_enable_reg_write, ls1_enable_reg_write, br1_enable_reg_write:

*   Enable register write flags for units

*   - p1_out, ls1_out, br1_out: Output data for units

```
*   - p1_addr_out, ls1_addr_out, br1_addr_out: Output register addresses for units
*   - p1_write_out, ls1_write_out, br1_write_out: Write flags for units
*****************************************************************************************************/

module Odd_Pipe_TB ();
  logic clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  logic [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  logic [ 2:0] instr_format;
  // Execution unit of instruction
  logic [ 1:0] unit;
  // Destination register address
  logic [ 0:6] dest_reg_addr;
  // Values of source registers
  logic [0:127] src_reg_a, src_reg_b, store_reg, store_reg_odd;
  // Immediate value, truncated based on instruction format
  logic [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  logic enable_reg_write;
  // Program counter from IF stage
  logic [7:0] program_counter_input;
  // Output value of Stage 7
  logic [0:127] wb_data;
  // Destination register for write back data
  logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  logic wb_enable_reg_write;
  // New program counter for branch
  logic [7:0] program_counter_wb;
  // Indicates whether a branch instruction is taken
  logic branch_is_taken;
  // Indicates if data is forwarded from the write-back stage
```

```
logic forwarded_data_wb;
// Indicates if address is forwarded from the write-back stage
logic forwarded_address_wb;
// Indicates if write flag is forwarded from the write-back stage
logic forwarded_write_flag_wb;
// Indicates if the system is in initial state
logic initial_;
// Indicates if branch operations are disabled
logic disable_branch;
// Indicates the delayed register address in Local Store Unit 1
logic delayed_ls1_register_address;
// Indicates if register write is delayed in Local Store Unit 1
logic delayed_ls1_register_write;
// Indicates the delayed register address in Permute Unit 1
logic delayed_p1_register_address;
// Indicates if register write is delayed in Permute Unit 1
logic delayed_p1_register_write;

Odd_Pipe dut (
    clock,
    reset,
    op_code,
    instr_format,
    unit,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    store_reg,
    imm_value,
    enable_reg_write,
    program_counter_input,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    program_counter_wb,
```

```
    branch_is_taken,
    forwarded_data_wb,
    forwarded_address_wb,
    forwarded_write_flag_wb,
    initial_,
    disable_branch,
    delayed_ls1_register_address,
    delayed_ls1_register_write,
    delayed_p1_register_address,
    delayed_p1_register_write
);

// Set the initial state of the clock to zero
initial clock = 0;

// Toggle the clock value every 5 time units to simulate oscillation
always begin
  #5 clock = ~clock;
  program_counter_input = program_counter_input + 1;
end

initial begin
  //*************************************************************************************************************
  //Permute_TB
  reset = 1;
  instr_format = 3'b000;
  // Set the opcode for the Shift Left Halfword (shlh) operation
  op_code = 11'b01010110100;
  // Set the destination register address to $r3
  dest_reg_addr = 7'b0000011;
  // Set the value of source register A, Halfwords: 16'h0010
  src_reg_a = 128'h5A7F38E2B0C4D69E18F2C86B5A7F38E2;
  // Set the value of source register B, Halfwords: 16'h0001
  src_reg_b = 128'h1E9D83CAB4267F0E2B8A64C11E9D83CA;
  imm_value = 0;
```

```
enable_reg_write = 1;
#6;
// At 11ns, disable the reset, enabling the unit
reset = 0;
@(posedge clock);
#1;
// Set the opcode for the No Operation (nop) instruction
op_code = 0;
@(posedge clock);
#1;
op_code   = 11'b01000000001;
src_reg_a = 128'hC7063E4F1B28D5A3F9A87F41C7063E4F;
@(posedge clock);
#1;
src_reg_a = 128'hEF4528AB1D3976F825B6D843EF4528AB;
@(posedge clock);
#1;
op_code   = 11'b00111011111;
src_reg_a = 128'h1A73BF5D68294E1BCA053B921A73BF5D;
@(posedge clock);
#1;
// Set the opcode for the "Shift Left Quadword by Bits Immediate" operation (shlqbii rt, src_reg_a, value)
op_code = 11'b00111111011;
// Set the destination register address to $r6
dest_reg_addr = 7'b0000110;
// Set the instruction format to RI7-type
instr_format = 2;
// Set the immediate value to 3
imm_value = 7'b0000011;
src_reg_a = 128'hB7D84231F6A905E378CD720EB7D84231;
@(posedge clock);
#4;
// Testcase for Shift Left Quadword by Bits (shlqbi rt, src_reg_a, src_reg_b)
src_reg_b = 128'h04987A6DB123FEDCFAE9080D04987A6D;
src_reg_a = 128'h96FC4D1E82AB0E64F3D89B2A96FC4D1E;
```

```
op_code = 11'b00111011011;
dest_reg_addr = 7'b0000101;
@(posedge clock);
#4;
// Set the opcode for the "Shift Left Quadword by Bits Immediate" operation (shlqbii rt, src_reg_a, value)
op_code = 11'b00111111011;
// Set the destination register address to $r6
dest_reg_addr = 7'b0000110;
// Set the instruction format to RI7-type
instr_format = 2;
// Set the immediate value to 3
imm_value = 7'b0000011;
src_reg_a = 128'h3E50187ABF6C9D24E6A3B59B3E50187A;
@(posedge clock);
#4;
// Testcase for Shift Left Quadword by Bytes (shlqby rt, src_reg_a, src_reg_b)
src_reg_b = 128'h8D7E20BC64F5A1C3E9D0B73F8D7E20BC;
src_reg_a = 128'hFAC825D7B39461E20E37A7B5FAC825D7;
op_code = 11'b00111011111;
dest_reg_addr = 7'b0000110;
@(posedge clock);
#4;
// Testcase for Shift Left Quadword by Bytes from Bit Shift Count (shlqbybi rt, src_reg_a, src_reg_b)
src_reg_b = 128'h0E6D9CAB724F158D37A109380E6D9CAB;
src_reg_a = 128'h29C673E10A4FEDB21B359A5629C673E1;
op_code = 11'b00111001111;
dest_reg_addr = 7'b0000111;
@(posedge clock);
#4;
// Testcase for Rotate Quadword by Bytes (rotqby rt, src_reg_a, src_reg_b)
src_reg_b = 128'hB804ED1AF5A69782CD43F0BEB804ED1A;
src_reg_a = 128'hA6E9D8013FCB24A857D0ECA3A6E9D801;
op_code = 11'b00111011100;
dest_reg_addr = 7'b0001000;
@(posedge clock);
```

```
#4;
// Testcase for Rotate Quadword by Bytes from Bit Shift Count (rotqbybi rt, src_reg_a, src_reg_b)
src_reg_b = 128'h2D18C9F6E3A5B470F1E862492D18C9F6;
src_reg_a = 128'hBEAF572C4D1E0F38C62B91E6BEAF572C;
op_code = 11'b00111001100;
dest_reg_addr = 7'b0001001;
@(posedge clock);
#4;
// Set the opcode for the "Rotate Quadword by Bytes Immediate" operation (rotqbyi rt, src_reg_a, imm7)
op_code = 11'b00111111100;
// Set the destination register address to $r6
dest_reg_addr = 7'b0000110;
// Set the instruction format to RI7-type
instr_format = 2;
// Set the immediate value to 3
imm_value = 7'b0000011;
src_reg_a = 128'h2A49E815D0F3CB0E7ABD3C9D2A49E815;
@(posedge clock);
#4;
op_code = 0;
// Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
#100;
op_code = 11'b00000000000;
//*************************************************************************************************************


//Local_Store_TB
reset = 1;
instr_format = 3'b000;
// Set the opcode for the Store Quadword (stqx) operation
op_code = 11'b00101000100;
// Set the destination register address to $r3
dest_reg_addr = 7'b0000011;
//Halfwords: 16'h0001
src_reg_a = 128'hF9EBA5CD6078C31425F79B42F9EBA5CD;
//Halfwords: 16'h0001
```

```
src_reg_b = 128'hD5B91E846F7A3CE590D2E4D9D5B91E84;
store_reg_odd = 128'h1F34BE0A6D8C92F7B5A19E1A1F34BE0A;
imm_value = 12;
enable_reg_write = 1;
#6;
// At 11ns, disable the reset, enabling the unit
reset = 0;
@(posedge clock);
#1;
// Load Quadword (d-form)
op_code = 8'b00110100;
instr_format = 4'b1100;
dest_reg_addr = 7'b0000011;
src_reg_a = 128'hC39A50EB8FD64B12A7EFC3BBC39A50EB;
imm_value = 7'b0000000;
@(posedge clock);
#1;
// Load Quadword (x-form)
op_code = 11'b00111000100;
instr_format = 4'b1101;
dest_reg_addr = 7'b0000011;
src_reg_a = 128'h7DC680A1EF59D438A1CDE2A27DC680A1;
src_reg_b = 128'h9C86514EBDA07F53A1E25DC69C86514E;
@(posedge clock);
#1;
// Load Quadword (a-form)
op_code = 9'b001100001;
instr_format = 4'b1110;
dest_reg_addr = 7'b0000011;
src_reg_a = 128'h9C13F6DB0E24A85CB6F9A2F69C13F6DB;
imm_value = 7'b0000000;
@(posedge clock);
#1;
// Load Quadword Instruction Relative (a-form)
op_code = 9'b001100111;
```

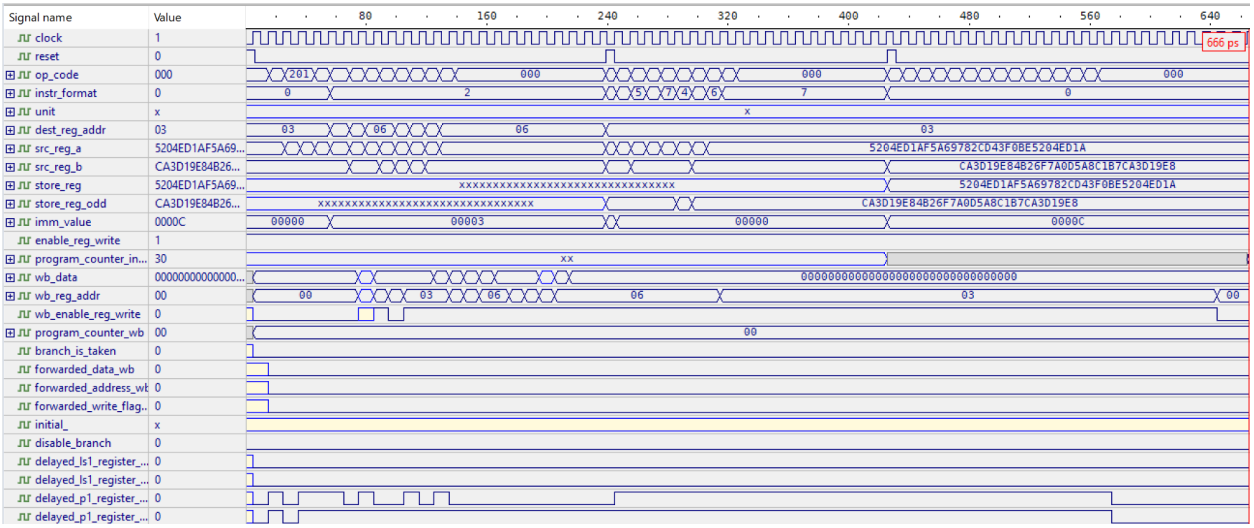
```
instr_format = 4'b1111;
dest_reg_addr = 7'b0000011;
src_reg_a = 128'hF6D5803C29B6E8C14A3F01D7F6D5803C;
imm_value = 7'b0000000;
@(posedge clock);
#1;
// Store Quadword (d-form)
op_code = 8'b00100100;
instr_format = 4'b1100;
src_reg_a = 128'h86E2A70D5F1D48CB9F3E0A5A86E2A70D;
store_reg_odd = 128'h8F7E20BC64F5A1C3E9D0B73F8F7E20BC;
imm_value = 7'b0000000;
@(posedge clock);
#1;
// Store Quadword (x-form)
op_code = 11'b00101000100;
instr_format = 4'b1101;
src_reg_a = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
src_reg_b = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
store_reg_odd = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
@(posedge clock);
#1;
// Store Quadword (a-form)
op_code = 9'b001000001;
instr_format = 4'b1110;
src_reg_a = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
store_reg_odd = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
imm_value = 7'b0000000;
@(posedge clock);
#1;
// Store Quadword Instruction Relative (a-form)
op_code = 9'b001000111;
instr_format = 4'b1111;
src_reg_a = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
store_reg_odd = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
```

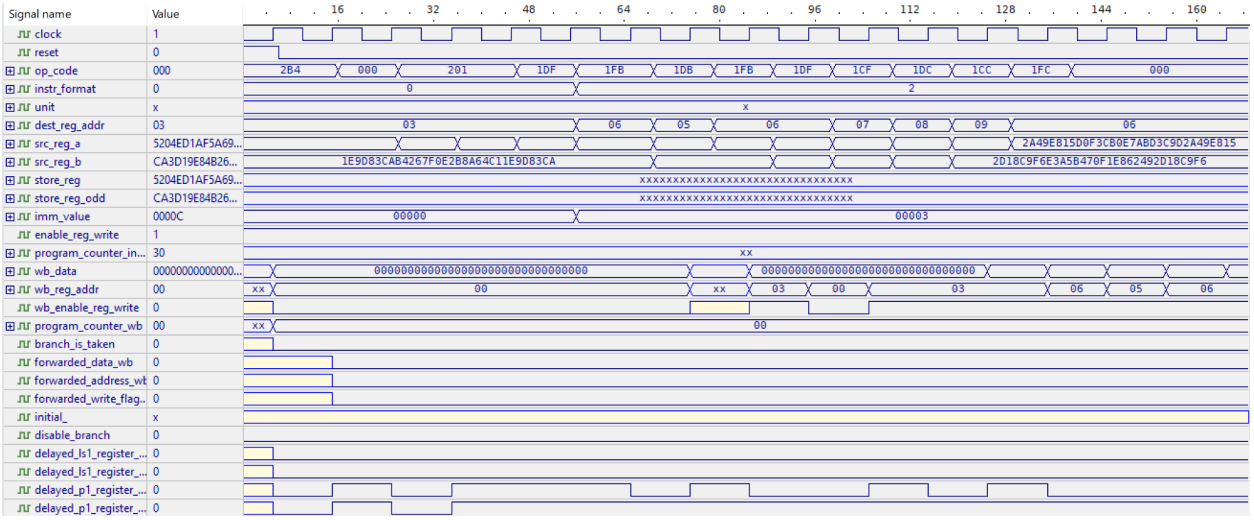
```
imm_value = 7'b0000000;
@(posedge clock);
#1;
// Set the opcode for the No Operation (nop) instruction
op_code = 0;
// Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
#100;
op_code = 11'b00000000000;
//*************************************************************************************************************


//Branch_TB
reset = 1;
instr_format = 3'b000;
// Set the opcode for the Branch Indirect instruction (bi)
op_code = 11'b00110101000;
// Set the destination register address to $r3
dest_reg_addr = 7'b0000011;
// Set the value of source register A
src_reg_a = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
// Set the value of source register B
src_reg_b = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
// Set the value of the store register
store_reg = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
imm_value = 12;
enable_reg_write = 1;
program_counter_input = 0;
#6;
// At 11ns, disable the reset, enabling the unit
reset = 0;
@(posedge clock);
#1;
// Branch Relative (br)
op_code = 9'b001100100;
@(posedge clock);
#1;
```

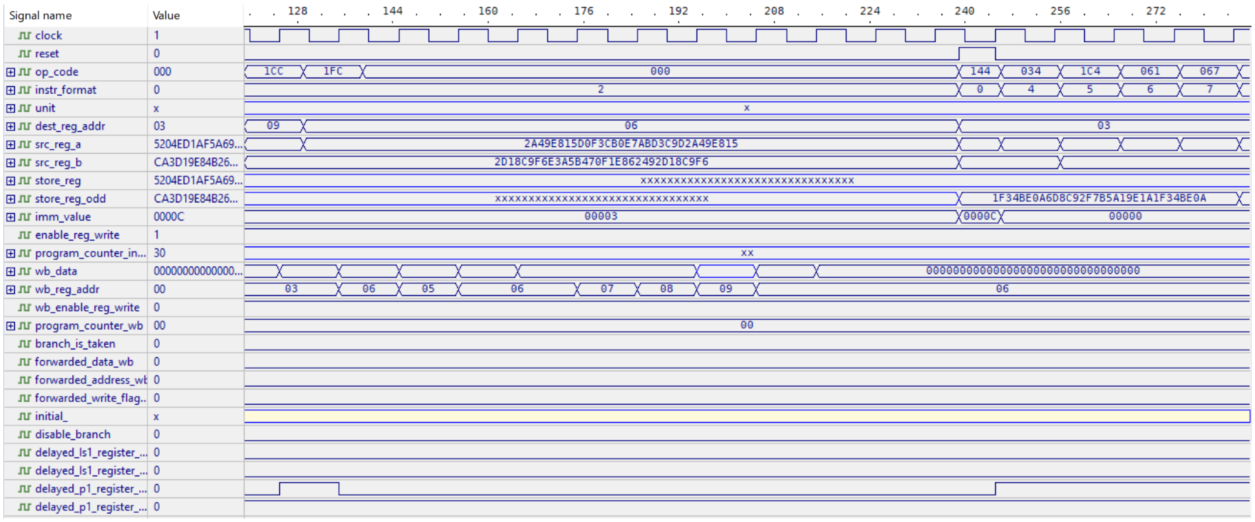
```
// Branch absolute (bra) operation
op_code = 9'b001100000;
@(posedge clock);
#1;
// Branch Relative and Set Link (brsl)
op_code = 9'b001100110;
@(posedge clock);
#1;
// Branch Absolute and Set Link (brasl)
op_code = 9'b001100010;
@(posedge clock);
#1;
// Branch Indirect (bi)
op_code = 11'b00110101000;
@(posedge clock);
#1;
// Branch If Not Zero Word (brnz)
op_code = 9'b001000010;
@(posedge clock);
#1;
// Branch If Zero Word (brz)
op_code = 9'b001000000;
@(posedge clock);
#1;
// Branch If Not Zero Halfword (brnzh)
op_code = 9'b001000110;
@(posedge clock);
#1;
// Branch If Zero Halfword (brzh)
op_code = 9'b001000100;
@(posedge clock);
#1;
// Branch Indirect If Not Zero (binz)
op_code = 11'b00100101001;
@(posedge clock);
```

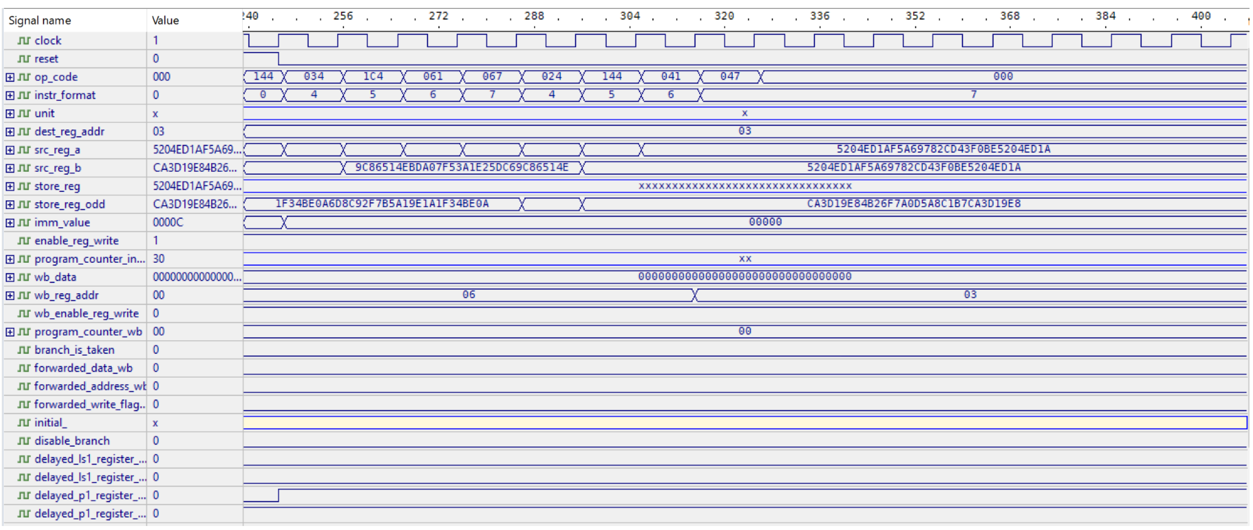
```
    #1;
    // Branch Indirect If Zero (biz)
    op_code = 11'b00100101000;
    @(posedge clock);
    #1;
    // Branch Indirect If Not Zero Halfword (binzh)
    op_code = 11'b00100101011;
    @(posedge clock);
    #1;
    // Branch Indirect If Zero Halfword (bihz)
    op_code = 11'b00100101010;
    @(posedge clock);
    #1;
    // Set the opcode for the No Operation (nop) instruction
    op_code = 0;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #100;
    op_code = 11'b00000000000;
    $stop;
  end
endmodule
```

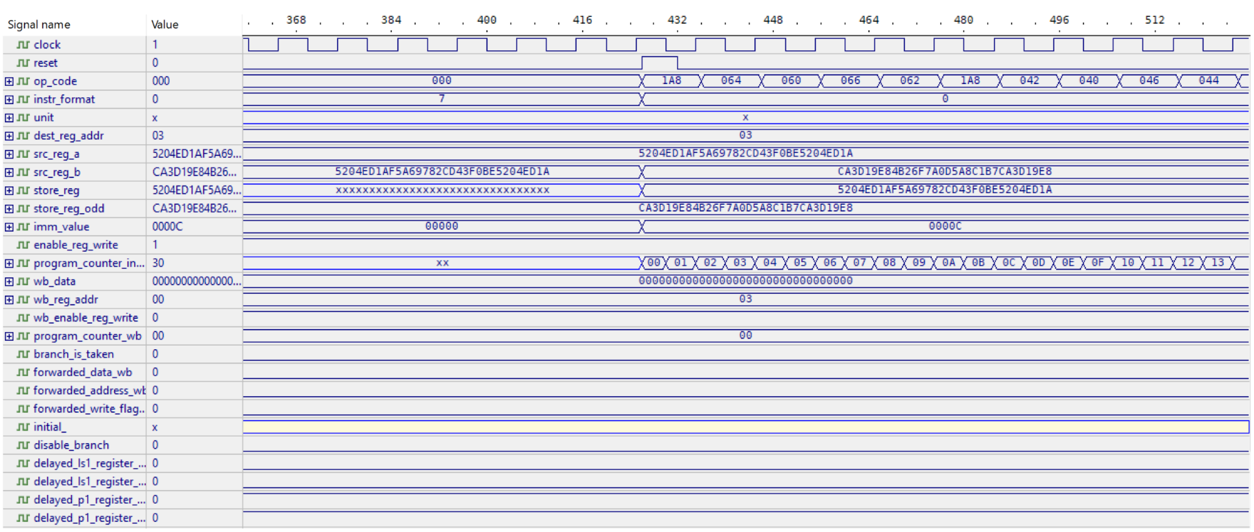
Odd Pipe Waveform:

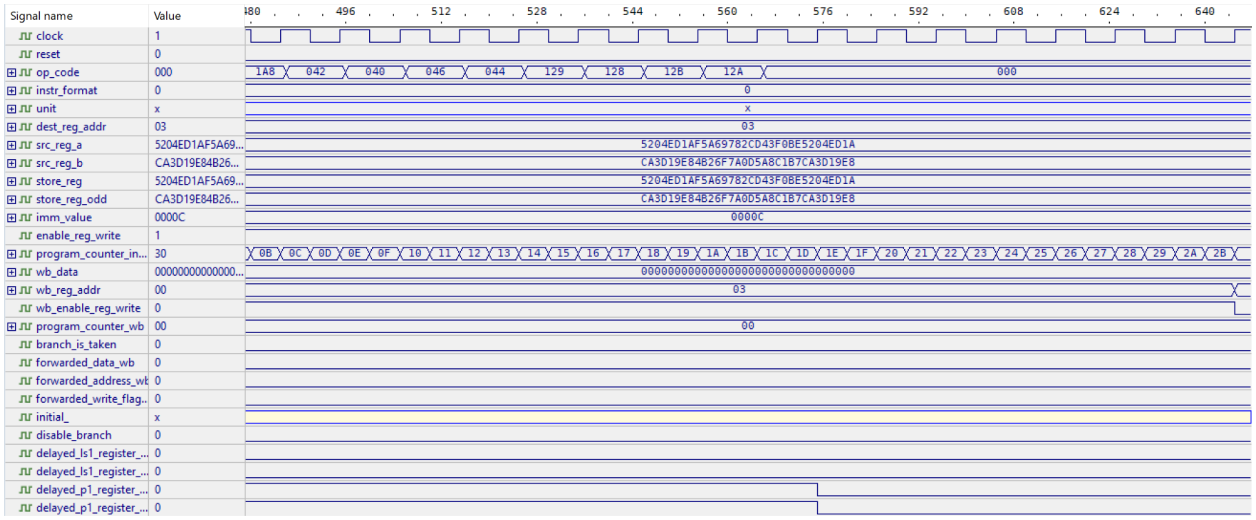


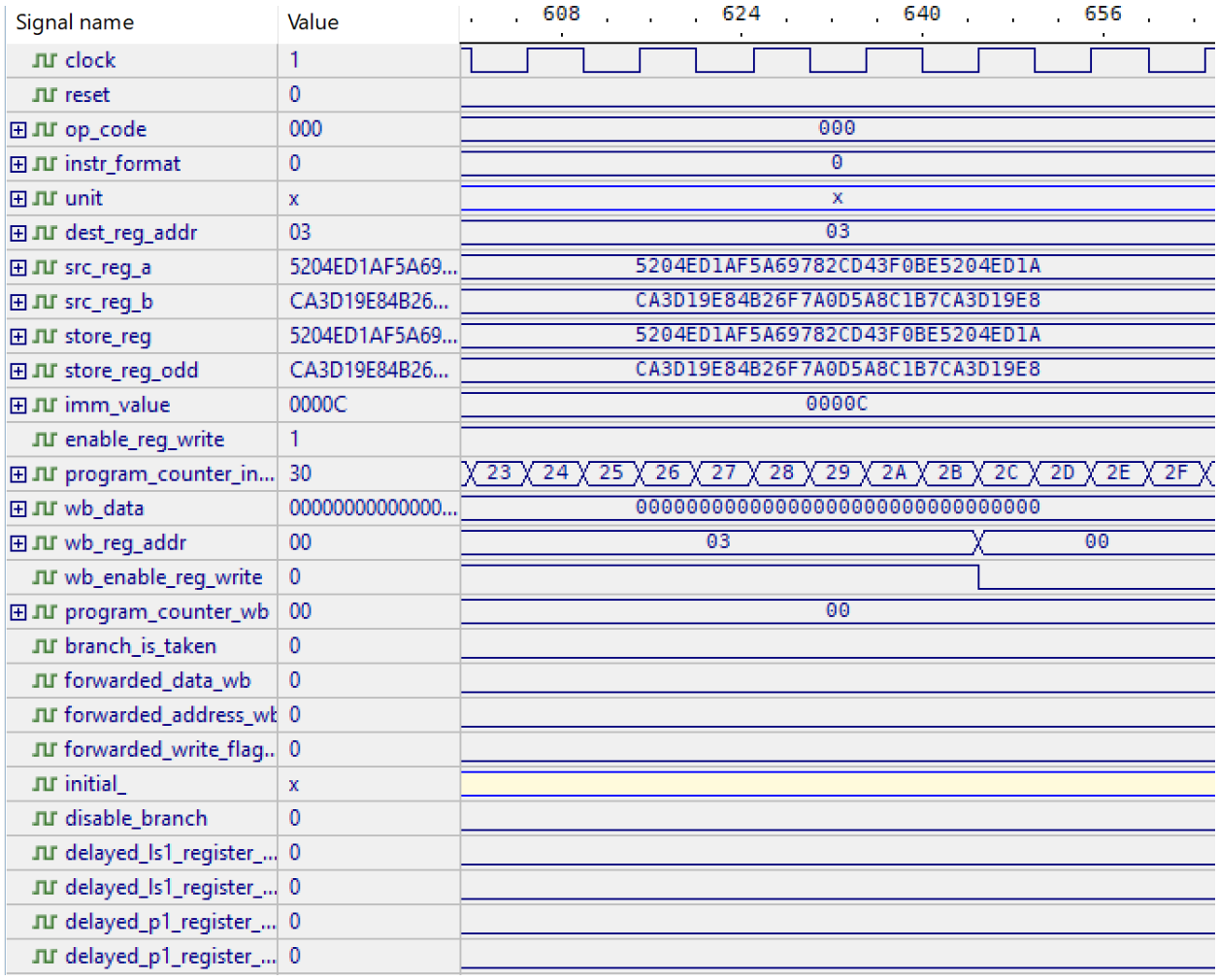



Signal name
Value
clock 1
reset 0
op_code 000
instr_format 0
unit x
dest_reg_addr 03
src_reg_a 5204ED1AF5A69...
src_reg_b CA3D19E84B26...
store_reg 5204ED1AF5A69...
store_reg_odd CA3D19E84B26...
imm_value 0000C
enable_reg_write 1
program_counter_in... 30
wb_data 00000000000000...
wb_reg_addr 00
wb_enable_reg_write 0
program_counter_wb 00
branch_is_taken 0
forwarded_data_wb 0
forwarded_address_wb 0
forwarded_write_flag.. 0
initial_ x
disable_branch 0
delayed_ls1_register_... 0
delayed_ls1_register_... 0
delayed_p1_register_... 0
delayed_p1_register_... 0


Signal name
Value
clock 1
reset 0
op_code 000
instr_format 0
unit x
dest_reg_addr 03
src_reg_a 5204ED1AF5A69...
src_reg_b CA3D19E84B26...
store_reg 5204ED1AF5A69...
store_reg_odd CA3D19E84B26...
imm_value 0000C
enable_reg_write 1
program_counter_in... 30
wb_data 00000000000000...
wb_reg_addr 00
wb_enable_reg_write 0
program_counter_wb 00
branch_is_taken 0
forwarded_data_wb 0
forwarded_address_wb 0
forwarded_write_flag.. 0
initial_ x
disable_branch 0
delayed_ls1_register_... 0
delayed_ls1_register_... 0
delayed_p1_register_... 0
delayed_p1_register_... 0


Signal name
Value
clock 1
reset 0
op_code 000
instr_format 0
unit x
dest_reg_addr 03
src_reg_a 5204ED1AF5A69...
src_reg_b CA3D19E84B26...
store_reg 5204ED1AF5A69...
store_reg_odd CA3D19E84B26...
imm_value 0000C
enable_reg_write 1
program_counter_in... 30
wb_data 00000000000000...
wb_reg_addr 00
wb_enable_reg_write 0
program_counter_wb 00
branch_is_taken 0
forwarded_data_wb 0
forwarded_address_wb 0
forwarded_write_flag.. 0
initial_ x
disable_branch 0
delayed_ls1_register_... 0
delayed_ls1_register_... 0
delayed_p1_register_... 0
delayed_p1_register_... 0


Signal name
Value
clock 1
reset 0
op_code 000
instr_format 0
unit x
dest_reg_addr 03
src_reg_a 5204ED1AF5A69...
src_reg_b CA3D19E84B26...
store_reg 5204ED1AF5A69...
store_reg_odd CA3D19E84B26...
imm_value 0000C
enable_reg_write 1
program_counter_in... 30
wb_data 00000000000000...
wb_reg_addr 00
wb_enable_reg_write 0
program_counter_wb 00
branch_is_taken 0
forwarded_data_wb 0
forwarded_address_wb 0
forwarded_write_flag.. 0
initial_ x
disable_branch 0
delayed_ls1_register_... 0
delayed_ls1_register_... 0
delayed_p1_register_... 0
delayed_p1_register_... 0


Signal name
Value
608
624
640
656
clock 1
reset 0
op_code 000
instr_format 0
unit x
dest_reg_addr 03
src_reg_a 5204ED1AF5A69... 5204ED1AF5A69782CD43F0BE5204ED1A
src_reg_b CA3D19E84B26... CA3D19E84B26F7A0D5A8C1B7CA3D19E8
store_reg 5204ED1AF5A69... 5204ED1AF5A69782CD43F0BE5204ED1A
store_reg_odd CA3D19E84B26... CA3D19E84B26F7A0D5A8C1B7CA3D19E8
imm_value 0000C
enable_reg_write 1
program_counter_in... 30
wb_data 00000000000000... 00000000000000000000000000000000
wb_reg_addr 00 03 00
wb_enable_reg_write 0
program_counter_wb 00
branch_is_taken 0
forwarded_data_wb 0
forwarded_address_wb 0
forwarded_write_flag.. 0
initial_ x
disable_branch 0
delayed_ls1_register_... 0
delayed_ls1_register_... 0
delayed_p1_register_... 0
delayed_p1_register_... 0

Even Pipe:

The “Even Pipe” module represents a crucial stage within a processor's pipeline architecture, specifically for the “even pipeline” stage. In a processor's pipeline, instructions go through several stages of processing before completing, with each stage handling different instruction executions. The Even Pipe module mainly manages instruction forwarding and handling, helping with the processor's pipeline.

The Even Pipe module takes in different inputs such as clock signals, reset signals, operation codes, instruction formats, source and destination register addresses, immediate values, and control flags. These inputs help to process incoming instructions and help to determine their execution paths within the processor. In this module it also generates outputs including data to be written back, register addresses, and flags indicating where a register write operation is enabled.

One important feature of the Even pipe modulus is its ability to handle read after write errors by implementing forwarding logic. This logic helps enable the module to detect dependencies between instruction and forward data from earlier pipeline stages to later stages, this helps avoid stalls and improve the overall performance of the pipeline. By using the forwarding paths and updating values based on the instruction execution, the module ensures that instructions progress through the pipeline without any data errors.

Overall, the Even Pipe module plays an important role in execution of instructions within a processor pipeline, managing instruction forwarding and handling in to optimize performance and lower stalls. This implementation helps demonstrate the key principles of pipeline processor design, including instroting flow control and error detection and mitigation.

Even Pipe Code:

```
/*************************************************************************************************************
 * Module: Even Pipe
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *     This module represents the even pipeline stage in a processor, handling instructions and forwarding
 *     logic.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
```

*   - op_code: Operation code for the instruction

*   - instr_format: Instruction format

*   - unit: Destination unit for the instruction

*   - dest_reg_addr: Destination register address

*   - src_reg_a: Value of source register A

*   - src_reg_b: Value of source register B

*   - temporary_register_c: Value of temporary register C

*   - imm_value: Immediate value

*   - enable_reg_write: Flag indicating whether register write is enabled

*   - branch_is_taken: Flag indicating whether branch is taken

 *----------

* Outputs:

*   - wb_data: Data to be written back

*   - wb_reg_addr: Address of the register to be written back

*   - wb_enable_reg_write: Flag indicating whether register write is enabled for write-back stage

*   - forwarded_data_wb: Forwarded data for write-back stage

*   - forwarded_address_wb: Forwarded address for write-back stage

*   - forwarded_write_flag_wb: Forwarded write flag for write-back stage

*   - delayed_rt_addr_fp1: Delayed register addresses for fp1 unit

*   - delayed_enable_reg_write_fp1: Delayed enable register write for fp1 unit

*   - delayed_int_fp1: Internal delay for fp1 unit

*   - delayed_rt_addr_fx2: Delayed register addresses for fx2 unit

*   - delayed_enable_reg_write_fx2: Delayed enable register write for fx2 unit

*   - delayed_rt_addr_b1: Delayed register addresses for b1 unit

*   - delayed_enable_reg_write_b1: Delayed enable register write for b1 unit

*   - delayed_rt_addr_fx1: Delayed register addresses for fx1 unit

*   - delayed_enable_reg_write_fx1: Delayed enable register write for fx1 unit

*----------

* Internal Signals:

*   - fp1_op_code, fx2_op_code, b1_op_code, fx1_op_code: Operation codes for units

*   - fp1_instruction_format, fx2_instruction_format, b1_instruction_format, fx1_instruction_format: Instruction formats for units

*   - fp1_enable_register_write, fx2_enable_register_write, b1_enable_register_write, fx1_enable_register_write:

*   Enable register write flags for units

*   - fp1_output_stage6, fx2_output_stage4, b1_output_stage4, fx1_output_stage2: Output data for units

*   - fp1_output_register, fx2_output_register, b1_output_register, fx1_output_register: Output register addresses for units

```
 *   - fp1_write_output, fx2_write_output, b1_write_output, fx1_write_output: Write flags for units
 *   - fp1_output_stage7: Internal data for fp1 unit
 *   - fp1_output_register_stage7: Internal register address for fp1 unit
 *   - fp1_write_output_stage7: Internal write flag for fp1 unit
 ****************************************************************************************************/

module Even_Pipe (
    clock,
    reset,
    op_code,
    instr_format,
    unit,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    temporary_register_c,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    forwarded_data_wb,
    forwarded_address_wb,
    forwarded_write_flag_wb,
    delayed_rt_addr_fp1,
    delayed_enable_reg_write_fp1,
    delayed_int_fp1,
    delayed_rt_addr_fx2,
    delayed_enable_reg_write_fx2,
    delayed_rt_addr_b1,
    delayed_enable_reg_write_b1,
    delayed_rt_addr_fx1,
    delayed_enable_reg_write_fx1
);
```

```
input clock, reset;

//Register File/Forwarding Stage
// Decoded opcode, truncated based on the instruction format
input [0:10] op_code;
// Format of the instruction, used in conjunction with op_code and imm_value
input [2:0] instr_format;
// Execution unit of the instruction
input [1:0] unit;
// Address of the destination register
input [0:6] dest_reg_addr;
// Values of source registers A, B, and temporary register C
input [0:127] src_reg_a, src_reg_b, temporary_register_c;
// Immediate value, truncated based on the instruction format
input [0:17] imm_value;
// Flag indicating if the current instruction will write to the Register Table
input enable_reg_write;
// Flag indicating if a branch was taken
input branch_is_taken;

// Write Back Stage
// Output value of Stage 7
output logic [0:127] wb_data;
// Address of the destination register for wb_data
output logic [0:6] wb_reg_addr;
// Flag indicating if wb_data will be written to the Register Table
output logic wb_enable_reg_write;

// Internal Signals
// Staging register for forwarded values
output logic [6:0][0:127] forwarded_data_wb;
// Address of the destination register for forwarded data
output logic [6:0][0:6] forwarded_address_wb;
// Flag indicating if forwarded data will be written to the Register Table
```

```
output logic [6:0] forwarded_write_flag_wb;


// Multiplexed opcode for the FP1 execution unit
logic [0:10] fp1_op_code;
// Multiplexed instruction format for the FP1 execution unit
logic [2:0] fp1_instruction_format;
// Flag indicating if the FP1 execution unit will write to the Register Table
logic fp1_enable_register_write;
// Output value of Stage 6 in the FP1 execution unit
logic [0:127] fp1_output_stage6;
// Destination register for the output data of the FP1 execution unit
logic [0:6] fp1_output_register;
// Flag indicating if the output data of the FP1 execution unit will be written to the Register Table
logic fp1_write_output;
// Output value of Stage 7 in the FP1 execution unit
logic [0:127] fp1_output_stage7;
// Destination register for the output data of Stage 7 in the FP1 execution unit
logic [0:6] fp1_output_register_stage7;
// Flag indicating if the output data of Stage 7 in the FP1 execution unit will be written to the Register Table
logic fp1_write_output_stage7;
// Multiplexed opcode for the FX2 execution unit
logic [0:10] fx2_op_code;
// Multiplexed instruction format for the FX2 execution unit
logic [2:0] fx2_instruction_format;
// Flag indicating if the FX2 execution unit will write to the Register Table
logic fx2_enable_register_write;
// Output value of Stage 4 in the FX2 execution unit
logic [0:127] fx2_output_stage4;
// Destination register for the output data of the FX2 execution unit
logic [0:6] fx2_output_register;
// Flag indicating if the output data of the FX2 execution unit will be written to the Register Table
logic fx2_write_output;
// Multiplexed opcode for the B1 execution unit
logic [0:10] b1_op_code;
// Multiplexed instruction format for the B1 execution unit
```

```
logic [2:0] b1_instruction_format;
// Flag indicating if the B1 execution unit will write to the Register Table
logic b1_enable_register_write;
// Output value of Stage 4 in the B1 execution unit
logic [0:127] b1_output_stage4;
// Destination register for the output data of the B1 execution unit
logic [0:6] b1_output_register;
// Flag indicating if the output data of the B1 execution unit will be written to the Register Table
logic b1_write_output;
// Multiplexed opcode for the FX1 execution unit
logic [0:10] fx1_op_code;
// Multiplexed instruction format for the FX1 execution unit
logic [2:0] fx1_instruction_format;
// Flag indicating if the FX1 execution unit will write to the Register Table
logic fx1_enable_register_write;
// Output value of Stage 2 in the FX1 execution unit
logic [0:127] fx1_output_stage2;
// Destination register for the output data of the FX1 execution unit
logic [0:6] fx1_output_register;
// Flag indicating if the output data of the FX1 execution unit will be written to the Register Table
logic fx1_write_output;

// Internal Signals for Handling Read-After-Write (RAW) Errors
// Delayed destination register address for the FP1 execution unit
output logic [6:0][0:6] delayed_rt_addr_fp1;
// Delayed flag indicating if the FP1 execution unit will write to the Register Table
output logic [6:0] delayed_enable_reg_write_fp1;
// Delayed flag indicating if the FP1 execution unit will write an integer result
output logic [6:0] delayed_int_fp1;
// Delayed destination register address for the FX2 execution unit
output logic [3:0][0:6] delayed_rt_addr_fx2;
// Delayed flag indicating if the FX2 execution unit will write to the Register Table
output logic [3:0] delayed_enable_reg_write_fx2;
// Delayed destination register address for the B1 execution unit
output logic [3:0][0:6] delayed_rt_addr_b1;
```

```
// Delayed flag indicating if the B1 execution unit will write to the Register Table
output logic [3:0] delayed_enable_reg_write_b1;
// Delayed destination register address for the FX1 execution unit
output logic [1:0][0:6] delayed_rt_addr_fx1;
// Delayed flag indicating if the FX1 execution unit will write to the Register Table
output logic [1:0] delayed_enable_reg_write_fx1;

Single_Precision fp1 (
    .clock(clock),
    .reset(reset),
    .op_code(fp1_op_code),
    .instr_format(fp1_instruction_format),
    .dest_reg_addr(dest_reg_addr),
    .src_reg_a(src_reg_a),
    .src_reg_b(src_reg_b),
    .temporary_register_c(temporary_register_c),
    .imm_value(imm_value),
    .enable_reg_write(fp1_enable_register_write),
    .wb_data(fp1_output_stage6),
    .wb_reg_addr(fp1_output_register),
    .wb_enable_reg_write(fp1_write_output),
    .int_data(fp1_output_stage7),
    .int_reg_addr(fp1_output_register_stage7),
    .int_enable_reg_write(fp1_write_output_stage7),
    .branch_is_taken(branch_is_taken),
    .delayed_rt_addr(delayed_rt_addr_fp1),
    .delayed_enable_reg_write(delayed_enable_reg_write_fp1),
    .int_operation_flag(delayed_int_fp1)
);
Simple_Fixed_2 fx2 (
    .clock(clock),
    .reset(reset),
    .op_code(fx2_op_code),
    .instr_format(fx2_instruction_format),
    .dest_reg_addr(dest_reg_addr),
```

```
    .src_reg_a(src_reg_a),
    .src_reg_b(src_reg_b),
    .imm_value(imm_value),
    .enable_reg_write(fx2_enable_register_write),
    .wb_data(fx2_output_stage4),
    .wb_reg_addr(fx2_output_register),
    .wb_enable_reg_write(fx2_write_output),
    .branch_is_taken(branch_is_taken),
    .delayed_rt_addr(delayed_rt_addr_fx2),
    .delayed_enable_reg_write(delayed_enable_reg_write_fx2)
);
Byte b1 (
    .clock(clock),
    .reset(reset),
    .op_code(b1_op_code),
    .instr_format(b1_instruction_format),
    .dest_reg_addr(dest_reg_addr),
    .src_reg_a(src_reg_a),
    .src_reg_b(src_reg_b),
    .imm_value(imm_value),
    .enable_reg_write(b1_enable_register_write),
    .wb_data(b1_output_stage4),
    .wb_reg_addr(b1_output_register),
    .wb_enable_reg_write(b1_write_output),
    .branch_is_taken(branch_is_taken),
    .delayed_rt_addr(delayed_rt_addr_b1),
    .delayed_enable_reg_write(delayed_enable_reg_write_b1)
);
Simple_Fixed_1 fx1 (
    .clock(clock),
    .reset(reset),
    .op_code(fx1_op_code),
    .instr_format(fx1_instruction_format),
    .dest_reg_addr(dest_reg_addr),
    .src_reg_a(src_reg_a),
```

```
    .src_reg_b(src_reg_b),
    .store_reg(temporary_register_c),
    .imm_value(imm_value),
    .enable_reg_write(fx1_enable_register_write),
    .wb_data(fx1_output_stage2),
    .wb_reg_addr(fx1_output_register),
    .wb_enable_reg_write(fx1_write_output),
    .branch_is_taken(branch_is_taken),
    .delayed_rt_addr(delayed_rt_addr_fx1),
    .delayed_enable_reg_write(delayed_enable_reg_write_fx1)
);

always_comb begin
  fp1_op_code = 0;
  fp1_instruction_format = 0;
  fp1_enable_register_write = 0;
  fx2_op_code = 0;
  fx2_instruction_format = 0;
  fx2_enable_register_write = 0;
  b1_op_code = 0;
  b1_instruction_format = 0;
  b1_enable_register_write = 0;
  fx1_op_code = 0;
  fx1_instruction_format = 0;
  fx1_enable_register_write = 0;

  // Multiplexer to determine which execution unit will process the instruction, based on the 'unit' input
  case (unit)
    // Case when the instruction is going to the Single Precision Unit 1 (FP1)
    2'b00: begin
      fp1_op_code = op_code;
      fp1_instruction_format = instr_format;
      fp1_enable_register_write = enable_reg_write;
    end
    // Case when the instruction is going to the Simple Fixed Point Unit 2 (FX2)
```

```
    2'b01: begin
      fx2_op_code = op_code;
      fx2_instruction_format = instr_format;
      fx2_enable_register_write = enable_reg_write;
    end
    // Case when the instruction is going to the Byte Unit 1 (B1)
    2'b10: begin
      b1_op_code = op_code;
      b1_instruction_format = instr_format;
      b1_enable_register_write = enable_reg_write;
    end
    // Case when the instruction is going to the Simple Fixed Point Unit 1 (FX1)
    2'b11: begin
      fx1_op_code = op_code;
      fx1_instruction_format = instr_format;
      fx1_enable_register_write = enable_reg_write;
    end
  endcase
end

always_ff @(posedge clock) begin
  // Initialize forwarding paths to 0
  for (int i = 0; i < 7; i++) begin
    forwarded_data_wb[i] <= 0;
    forwarded_address_wb[i] <= 0;
    forwarded_write_flag_wb[i] <= 0;
  end
  if (reset == 1) begin
    // Reset values
    wb_data <= 0;
    wb_reg_addr <= 0;
    wb_enable_reg_write <= 0;
  end else begin
    // Update values based on forwarding paths
    wb_data <= forwarded_data_wb[6];
```

```
    wb_reg_addr <= forwarded_address_wb[6];
    wb_enable_reg_write <= forwarded_write_flag_wb[6];

    // Forwarding paths
    if (fp1_write_output_stage7 == 1) begin
      forwarded_data_wb[6] <= fp1_output_stage7;
      forwarded_address_wb[6] <= fp1_output_register_stage7;
      forwarded_write_flag_wb[6] <= fp1_write_output_stage7;
    end else if (fp1_write_output == 1) begin
      forwarded_data_wb[5] <= fp1_output_stage6;
      forwarded_address_wb[5] <= fp1_output_register;
      forwarded_write_flag_wb[5] <= fp1_write_output;
    end else if (fx2_write_output == 1) begin
      forwarded_data_wb[3] <= fx2_output_stage4;
      forwarded_address_wb[3] <= fx2_output_register;
      forwarded_write_flag_wb[3] <= fx2_write_output;
    end else if (b1_write_output == 1) begin
      forwarded_data_wb[3] <= b1_output_stage4;
      forwarded_address_wb[3] <= b1_output_register;
      forwarded_write_flag_wb[3] <= b1_write_output;
    end else if (fx1_write_output == 1) begin
      forwarded_data_wb[1] <= fx1_output_stage2;
      forwarded_address_wb[1] <= fx1_output_register;
      forwarded_write_flag_wb[1] <= fx1_write_output;
    end
    // Propagate values through forwarding paths
    for (int i = 6; i > 0; i = i - 1) begin
      forwarded_data_wb[i-1] <= forwarded_data_wb[i];
      forwarded_address_wb[i-1] <= forwarded_address_wb[i];
      forwarded_write_flag_wb[i-1] <= forwarded_write_flag_wb[i];
    end
   end
  end
endmodule
```

Even Pipe Test Bench:

The Even Pipe testbench is an important component for verifying the functionality of the Even Pipe model, which constitutes a significant part of a processor's pipeline architecture. It serves as the stage where instructions are processed and forwarded through. The module operates based on the clock signal and a reset signal, which helps to ensure synchronization and initialization.

The inputs to the Even Pipe testbench includes various control signals and data associated with each instruction, such as operation code, instruction format, destination unit, source register values, immediate values, and flags indicating register write enable and branch prediction. These inputs are important to simulate the behavior of the Even Pipe modules under different instructions.

Outputs from the test bench include data and control signals related to the write back stage, such as data to be written back, destination of the register address, and even flags indicating register write enable for the write back stage. Additionally, forwarded data and address signals for the write back stage are provided, along with delayed register addresses and enable register write signals for various execution units. By defining these inputs, outputs, and internal signals, the testbench ensures verification of the Even Pipe modules functionality, contributing to the overall performance and reliability of the processors pipeline architecture.

Even Pipe Test Bench Code:

```
/*****************************************************************************************************
 * Module: Even Pipe Testbench
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------
 * Description:
 *     This testbench module verifies the functionality of the Even Pipe module, which represents the even
 *     pipeline stage in a processor, handling instructions and forwarding logic.
 *-----------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Operation code for the instruction
 *   - instr_format: Instruction format
 *   - unit: Destination unit for the instruction
 *   - dest_reg_addr: Destination register address
```

*   - src_reg_a: Value of source register A

*   - src_reg_b: Value of source register B

*   - temporary_register_c: Value of temporary register C

*   - imm_value: Immediate value

*   - enable_reg_write: Flag indicating whether register write is enabled

*   - branch_is_taken: Flag indicating whether branch is taken

 *---

* Outputs:

*   - wb_data: Data to be written back

*   - wb_reg_addr: Address of the register to be written back

*   - wb_enable_reg_write: Flag indicating whether register write is enabled for write-back stage

*   - forwarded_data_wb: Forwarded data for write-back stage

*   - forwarded_address_wb: Forwarded address for write-back stage

*   - forwarded_write_flag_wb: Forwarded write flag for write-back stage

*   - delayed_rt_addr_fp1: Delayed register addresses for fp1 unit

*   - delayed_enable_reg_write_fp1: Delayed enable register write for fp1 unit

*   - delayed_int_fp1: Internal delay for fp1 unit

*   - delayed_rt_addr_fx2: Delayed register addresses for fx2 unit

*   - delayed_enable_reg_write_fx2: Delayed enable register write for fx2 unit

*   - delayed_rt_addr_b1: Delayed register addresses for b1 unit

*   - delayed_enable_reg_write_b1: Delayed enable register write for b1 unit

*   - delayed_rt_addr_fx1: Delayed register addresses for fx1 unit

*   - delayed_enable_reg_write_fx1: Delayed enable register write for fx1 unit

*---

* Internal Signals:

*   - fp1_op_code, fx2_op_code, b1_op_code, fx1_op_code: Operation codes for units

*   - fp1_instr_format, fx2_instr_format, b1_instr_format, fx1_instr_format: Instruction formats for units

*   - fp1_enable_reg_write, fx2_enable_reg_write, b1_enable_reg_write, fx1_enable_reg_write:

*   Enable register write flags for units

*   - fp1_out, fx2_out, b1_out, fx1_out: Output data for units

*   - fp1_addr_out, fx2_addr_out, b1_addr_out, fx1_addr_out: Output register addresses for units

*   - fp1_write_out, fx2_write_out, b1_write_out, fx1_write_out: Write flags for units

*   - fp1_int: Internal data for fp1 unit

*   - fp1_addr_int: Internal register address for fp1 unit

*   - fp1_write_int: Internal write flag for fp1 unit

```
**********************************************************************************************************/

module Even_Pipe_TB ();
  logic clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  logic [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  logic [ 2:0] instr_format;
  // Execution unit of instruction
  logic [ 1:0] unit;
  // Destination register address
  logic [ 0:6] dest_reg_addr;
  // Values of source registers
  logic [0:127] src_reg_a, src_reg_b, temporary_register_c, store_reg;
  // Immediate value, truncated based on instruction format
  logic [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  logic enable_reg_write;

  // Write Back Stage
  // Output value of Stage 7
  logic [0:127] wb_data;
  // Destination register for write back data
  logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  logic wb_enable_reg_write;
  // Indicates whether a branch instruction is taken
  logic branch_is_taken;
  // Indicates if data is forwarded from the write-back stage
  logic forwarded_data_wb;
  // Indicates if address is forwarded from the write-back stage
  logic forwarded_address_wb;
  // Indicates if write flag is forwarded from the write-back stage
```

```
logic forwarded_write_flag_wb;
// Indicates the delayed register address in the Single Precision Unit 1
logic delayed_rt_addr_fp1;
// Indicates if register write is delayed in Single Precision Unit 1
logic delayed_enable_reg_write_fp1;
// Indicates the delayed interrupt in the Single Precision Unit 1
logic delayed_int_fp1;
// Indicates the delayed register address in the Simple Fixed 2 Unit 2
logic delayed_rt_addr_fx2;
// Indicates if register write is delayed in the Simple Fixed 2 Unit 2
logic delayed_enable_reg_write_fx2;
// Indicates the delayed register address in Byte Unit 1
logic delayed_rt_addr_b1;
// Indicates if register write is delayed in Byte Unit 1
logic delayed_enable_reg_write_b1;
// Indicates the delayed register address in the Simple Fixed 1 Unit 1
logic delayed_rt_addr_fx1;
// Indicates if register write is delayed in the Simple Fixed 1 Unit 1
logic delayed_enable_reg_write_fx1;

Even_Pipe dut (
    clock,
    reset,
    op_code,
    instr_format,
    unit,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    temporary_register_c,
    imm_value,
    enable_reg_write,
    wb_data,
    branch_is_taken,
    forwarded_data_wb,
```

```
    forwarded_address_wb,
    forwarded_write_flag_wb,
    delayed_rt_addr_fp1,
    delayed_enable_reg_write_fp1,
    delayed_int_fp1,
    delayed_rt_addr_fx2,
    delayed_enable_reg_write_fx2,
    delayed_rt_addr_b1,
    delayed_enable_reg_write_b1,
    delayed_rt_addr_fx1,
    delayed_enable_reg_write_fx1,
    wb_reg_addr,
    wb_enable_reg_write
);

// Set the initial state of the clock to zero
initial clock = 0;

// Oscillate the clock: every 5 time units, it changes its value
always begin
  #5 clock = ~clock;
end

initial begin
  //*****************************************************************************************************************
  //Single_Precision_TB
  reset = 1;
  instr_format = 3'b000;
  // Multiply
  op_code = 11'b01111000100;
  dest_reg_addr = 7'b0000011;
  src_reg_a = 128'h3F1A6D9E42B7C8F1A1E2D3F4A5B6C7D8;
  src_reg_b = 128'h9B8C7D6E5F4A3B2C1D2E3F4A5B6C7D8;
  temporary_register_c = 128'hF5E4D3C2B1A09876543210ABCDEF012;
  imm_value = 42;
```

```
enable_reg_write = 1;
#6;
reset = 0;
@(posedge clock);
#8;
// Multiply Unsigned
op_code = 11'b01111001100;
@(posedge clock);
#8;
// Multiply Immediate
op_code = 8'b01110100;
@(posedge clock);
#8;
// Multiply Unsigned Immediate
op_code = 8'b01110101;
@(posedge clock);
#8;
// Multiply and Add
op_code = 4'b1100;
@(posedge clock);
#8;
// Multiply High
op_code = 11'b01111000101;
@(posedge clock);
#8;
// Multiply and Shift Right
op_code = 11'b01111000111;
@(posedge clock);
#8;
// Multiply High High
op_code = 11'b01111000110;
@(posedge clock);
#8;
// Floating Add
op_code = 11'b01011000100;
```

```
dest_reg_addr = 7'b0000011;
src_reg_a = 128'hABCDEF0123456789ABCDEF012345678;
src_reg_b = 128'hFEDCBA9876543210FEDCBA987654321;
temporary_register_c = 128'h0123456789ABCDEF0123456789ABCDEF;
imm_value = 173;
@(posedge clock);
#8;
// Floating Subtract
op_code = 11'b01011000101;
@(posedge clock);
#8;
// Floating Multiply
op_code = 11'b01011000110;
@(posedge clock);
#8;
// Floating Multiply and Add
op_code = 4'b1110;
@(posedge clock);
#8;
// Floating Negative Multiply and Subtract
op_code = 4'b1101;
@(posedge clock);
#8;
// Floating Multiply and Subtract
op_code = 4'b1111;
@(posedge clock);
#8;
// Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
#200;
op_code = 11'b00000000000;
//*************************************************************************************************************

//Simple_Fixed_2_TB
reset = 1;
instr_format = 3'b000;
```

```
// Set the opcode for the Shift Left Halfword (shlh) operation
op_code = 11'b00001011111;
// Set the destination register address to $r3
dest_reg_addr = 7'b0000011;
// Set the value of source register A, Halfwords: 16'h0010
src_reg_a = 128'h7F8A9BACBDBECFD0E1F2030405060708;
// Set the value of source register B, Halfwords: 16'h0001
src_reg_b = 128'h0123456789ABCDEFABCDEF012345678;
imm_value = 0;
enable_reg_write = 1;
#6;
// At 11ns, disable the reset, enabling the unit
reset = 0;
@(posedge clock);
#1;
// Set the opcode for the No Operation (nop) instruction
op_code = 0;
@(posedge clock);
#1;
op_code = 11'b01000000001;
@(posedge clock);
#1;
op_code = 0;
@(posedge clock);
#1;
op_code = 11'b01000000001;
@(posedge clock);
#1;
op_code = 0;
@(posedge clock);
#1;
op_code = 11'b01000000001;
@(posedge clock);
#1;
// Handling the "Shift Left Word" operation (shl)
```

```
op_code   = 11'b00001011011;
src_reg_b = 128'hFEDCBA9876543210FEDCBA987654321;
src_reg_a = 128'hA1B2C3D4E5F67890A1B2C3D4E5F6789;
@(posedge clock);
#4;
// Handling the "Rotate Halfword" operation (roth)
op_code   = 11'b00001011100;
src_reg_b = 128'h1234567890ABCDEF1234567890ABCDEF;
src_reg_a = 128'hFEDCBA0987654321FEDCBA098765432;
@(posedge clock);
#4;
// Handling the "Rotate Word" operation (rot)
op_code   = 11'b00001011000;
src_reg_b = 128'h9876543210ABCDEF0123456789ABCDEF;
src_reg_a = 128'hFEDCBA987654321001234567 89ABCDEF;
@(posedge clock);
#4;
// Handling the "Rotate Word" operation (rot)
op_code   = 11'b00001011000;
src_reg_b = 128'hABCDEFFEDCBA9876543210ABCDEF012;
src_reg_a = 128'h0123456789ABCDEFABCDEF012345678;
@(posedge clock);
#4;
// Shift Left Halfword
op_code   = 11'b00001011111;
src_reg_b = 128'hFEDCBA9876543210FEDCBA9876543210;
src_reg_a = 128'h13579BDF2468ACE02468ACE13579BDF;
@(posedge clock);
#4;
// Shift Left Halfword Immediate
op_code = 11'b00001111111;
instr_format = 3'b010;
imm_value = 5;
src_reg_b = 128'hFEDCBA9876543210ABCDEF012345678;
src_reg_a = 128'h0123456789ABCDEFABCDEF0123456789;
```

```
@(posedge clock);
#4;
// Shift Left Word
op_code   = 11'b00001011011;
src_reg_b = 128'hFEDCBA9876543210FEDCBA987654321;
src_reg_a = 128'hABCDEFFEDCBA9876543210ABCDEFABC;
@(posedge clock);
#4;
// Shift Left Word Immediate
op_code = 11'b00001111011;
instr_format = 3'b010;
imm_value = 5;
src_reg_b = 128'h5A23F8F5A0CDE28F50E78A5A23F8F5A0;
src_reg_a = 128'h3B7912D6EFD84AB41021A683B7912D6E;
@(posedge clock);
#4;
// Rotate Word
op_code   = 11'b00001011000;
src_reg_b = 128'h8EBA1945D1C7F26378EBA1945D1C7F26;
src_reg_a = 128'hF4C65329B71D890F9DC17E8F4C65329B;
@(posedge clock);
#4;
// Rotate Word Immediate
op_code = 11'b00001111000;
instr_format = 3'b010;
imm_value = 5;
src_reg_b = 128'hE2A719385F2B91CDE89E2A719385F2B9;
src_reg_a = 128'h620DF4EAC1723B856BCA55B620DF4EAC;
@(posedge clock);
#4;
// Rotate Halfword
op_code   = 11'b00001011100;
src_reg_b = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
src_reg_a = 128'h478D6AC01257E3FBA7C3DEA478D6AC01;
@(posedge clock);
```

```
#4;
// Rotate Halfword Immediate
op_code = 11'b00001111100;
instr_format = 3'b010;
imm_value = 5;
src_reg_b = 128'hC1E9B54AD6F30E271BDE5A1C1E9B54AD;
src_reg_a = 128'h3C2FA791D68BFE92EDD6B7B3C2FA791D;
@(posedge clock);
#4;
// Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
#100;
op_code = 11'b00000000000;
//*************************************************************************************************************


//Byte_TB
reset = 1;
instr_format = 3'b000;
// Set the opcode for the Count Ones in Bytes (cntb) operation
op_code = 11'b01010110100;
// Set the destination register address to $r3
dest_reg_addr = 7'b0000011;
// Set the value of source register A,  Halfwords: 16'h0010
src_reg_a = 128'hABCDEF1234567890ABCDEF123456789;
// Set the value of source register B, Halfwords: 16'h0001
src_reg_b = 128'hFEDCBA0987654321FEDCBA098765432;
imm_value = 0;
enable_reg_write = 1;
#6;
// At 11ns, disable the reset, enabling the unit
reset = 0;
@(posedge clock);
#4;
// Set the opcode for the Average Bytes (avgb) operation
op_code = 11'b00011010011;
@(posedge clock);
```

```
#4;
// Set the opcode for the Absolute Differences of Bytes (absdb) operation
op_code = 11'b00001010011;
@(posedge clock);
#4;
// Set the opcode for the Sum Bytes into Halfwords (sumb) operation
op_code = 11'b01001010011;
@(posedge clock);
#4;
// Set the opcode for the No Operation (nop) instruction
op_code = 0;
// Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
#100;
op_code = 11'b00000000000;
//*****************************************************************************************************************

//Simple_Fixed_1_TB
reset = 1;
instr_format = 3'b000;
// Set the opcode for the Shift Left Halfword (shlh) operation
op_code = 11'b00001011111;
// Set the destination register address to $r3
dest_reg_addr = 7'b0000011;
// Set the value of source register A, Halfwords: 16'h0010
src_reg_a = 128'h1A2B3C4D5E6F7A8B9C0D1E2F3A4B5C6;
// Set the value of source register B, Halfwords: 16'h0001
src_reg_b = 128'hF0E1D2C3B4A59687766554433221100;
imm_value = 0;
enable_reg_write = 1;
#6;
// At 11ns, disable the reset, enabling the unit
reset = 0;
@(posedge clock);
#1;
// Set the opcode for the No Operation (nop) instruction
```

```
op_code = 0;
@(posedge clock);
// Add Extended
#3;
op_code   = 11'b01101000000;
src_reg_a = 128'h8C15F2E6A90DC4BF2A7899E8C15F2E6A;
src_reg_b = 128'h9AF507D38A4E62C711B7D459AF507D38;
@(posedge clock);
// Carry Generate
#3;
op_code   = 11'b00011000010;
src_reg_a = 128'h56187F3ED26A950DBF2AC3C56187F3ED;
src_reg_b = 128'hB6A9C81E57D9AF32F5E483BB6A9C81E5;
@(posedge clock);
// Subtract from Extended
#3;
op_code   = 11'b01101000001;
src_reg_a = 128'h6D475C3A1809E6ABDC30E126D475C3A1;
src_reg_b = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
@(posedge clock);
// Borrow Generate
#3;
op_code   = 11'b00001000010;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
src_reg_b = 128'h8F6D3BA24C7159FACF3F08C8F6D3BA24;
@(posedge clock);
// Add Halfword
#3;
op_code   = 11'b00011001000;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
src_reg_b = 128'h00000000000000000000000000000001;
@(posedge clock);
// Add Halfword
src_reg_a = 128'h9AF507D38A4E62C711B7D459AF507D38;
src_reg_b = 128'h0A21DECB7EF548D1D36E7860A21DECB7;
```

```
@(posedge clock);
#3;
op_code   = 11'b00011001000;
// Add Halfword
src_reg_a = 128'h849F36E1D4B5A2FC94DC931849F36E1D;
src_reg_b = 128'hE58179AFDB023865EEABE3DE58179AFD;
@(posedge clock);
#3;
src_reg_a = 128'h6A81F2CD45B796EDC7B2E996A81F2CD4;
@(posedge clock);
//sfh rt, ra, rb : Subtract from Halfword
#3;
op_code   = 11'b00001001000;
src_reg_a = 128'h478D6AC01257E3FBA7C3DEA478D6AC01;
src_reg_b = 128'h3B7912D6EFD84AB41021A683B7912D6E;
@(posedge clock);
//sfh rt, ra, rb : Subtract from Halfword
#3;
op_code   = 11'b00001001000;
src_reg_a = 128'hFD45F8F5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h8EBA1945D1C7F26378EBA1945D1C7F26;
@(posedge clock);
//sf rt, ra, rb : Subtract from Word
#3;
op_code   = 11'b00001000000;
src_reg_a = 128'h6D475C3A1809E6ABDC30E126D475C3A1;
src_reg_b = 128'hDFA5C9B3824FA71DB7B2EC5DFA5C9B38;
@(posedge clock);
//sf rt, ra, rb : Subtract from Word
#3;
op_code   = 11'b00001000000;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
src_reg_b = 128'h91BE7CAE5F34D2A4DB1E38C91BE7CAE5;
@(posedge clock);
//sfhi rt, ra, imm10 : Subtract from Halfword Immediate
```

```
#3;
op_code = 8'b00001101;
instr_format = 4;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
imm_value = 10'b0011111111;
@(posedge clock);
//sfhi rt, ra, imm10 : Subtract from Halfword Immediate
#3;
op_code = 8'b00001101;
instr_format = 4;
src_reg_a = 128'h0F3A9C15E6D7A8C34C27C930F3A9C15E;
imm_value = 10'b1011111111;
@(posedge clock);
//sfi rt, ra, imm10 : Subtract from Word Immediate
#3;
op_code = 8'b00001100;
instr_format = 4;
src_reg_a = 128'h4228000040647AE1BFC00000BB83126F;
imm_value = 10'b0011111111;
@(posedge clock);
//sfi rt, ra, imm10 : Subtract from Word Immediate
#3;
op_code = 8'b00001100;
src_reg_a = 128'h6246BA3C1485D99AC471226246BA3C14;
imm_value = 10'b1011111111;
instr_format = 4;
@(posedge clock);
//Add Word
#3;
op_code = 11'b00011000000;
instr_format = 0;
src_reg_a = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h3C2FA791D68BFE92EDD6B7B3C2FA791D;
@(posedge clock);
//Add Word
```

```
#3;
op_code   = 11'b00011000000;
src_reg_a = 128'h0A21DECB7EF548D1D36E7860A21DECB7;
src_reg_b = 128'h849F36E1D4B5A2FC94DC931849F36E1D;
@(posedge clock);
//ahi rt, ra, imm10 : Add Halfword Immediate
#3;
op_code = 8'b00011101;
instr_format = 4;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
imm_value = 10'b0011111111;
@(posedge clock);
//ai rt, ra, imm10 : Add Word Immediate
#3;
op_code = 8'b00011100;
instr_format = 4;
src_reg_a = 128'h8F6D3BA24C7159FACF3F08C8F6D3BA24;
imm_value = 10'b1011111111;
@(posedge clock);
instr_format = 3'b000;
// Handling the "AND" operation (and)
#3;
op_code   = 11'b00011000001;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
src_reg_b = 128'h3F87EDD295C61BAE46EF23B3F87EDD29;
@(posedge clock);
// Handling the "OR" operation (or)
#3;
op_code   = 11'b00001000001;
src_reg_a = 128'h4228000040647AE1BFC00000BB83126F;
src_reg_b = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
@(posedge clock);
// Handling the "XOR" operation (xor)
#3;
op_code = 11'b01001000001;
```

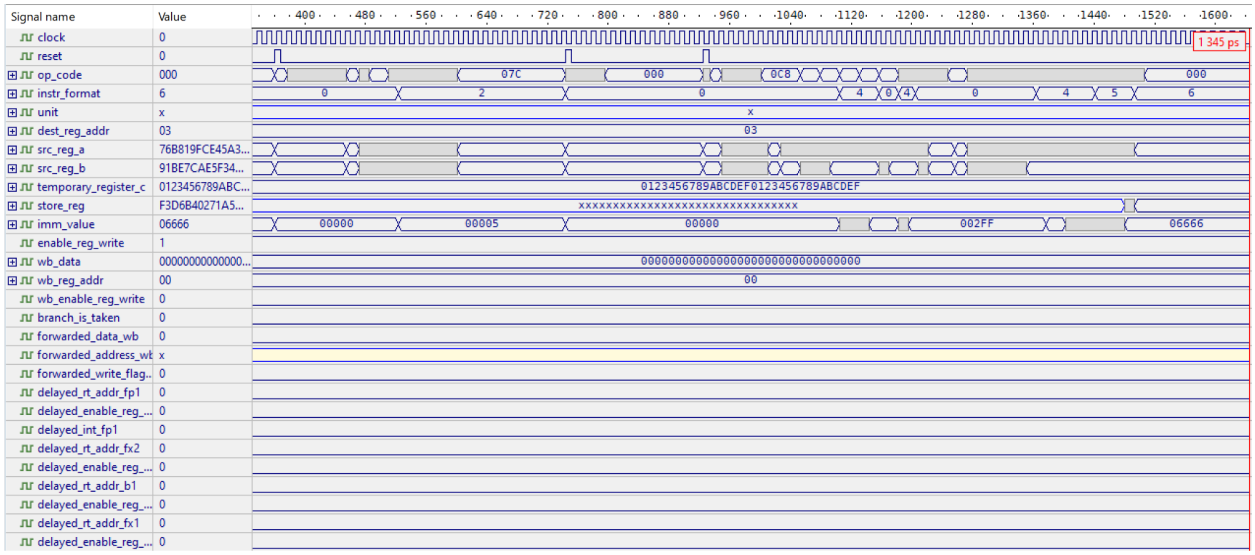
```
@(posedge clock);
// Handling the "NAND" operation (nand)
#3;
op_code = 11'b00011001001;
@(posedge clock);
src_reg_a = 128'hE2A719385F2B91CDE89E2A719385F2B9;
src_reg_b = 128'h91BE7CAE5F34D2A4DB1E38C91BE7CAE5;
@(posedge clock);
// ceqh rt, ra, rb Compare Equal Halfword
#3;
op_code   = 11'b01111001000;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
src_reg_b = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
@(posedge clock);
// ceq rt, ra, rb Compare Equal Word
#3;
op_code   = 11'b01111000000;
src_reg_a = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h8C15F2E6A90DC4BF2A7899E8C15F2E6A;
@(posedge clock);
// cgth rt, ra, rb Compare Greater Than Halfword
#3;
op_code   = 11'b01001001000;
src_reg_a = 128'h6246BA3C1485D99AC471226246BA3C14;
src_reg_b = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
@(posedge clock);
// cgt rt, ra, rb Compare Greater Than Word
#3;
op_code   = 11'b01001000000;
src_reg_a = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h91BE7CAE5F34D2A4DB1E38C91BE7CAE5;
@(posedge clock);
// clgtb rt, ra, rb Compare Logical Greater Than Byte
#3;
op_code   = 11'b01011010000;
```

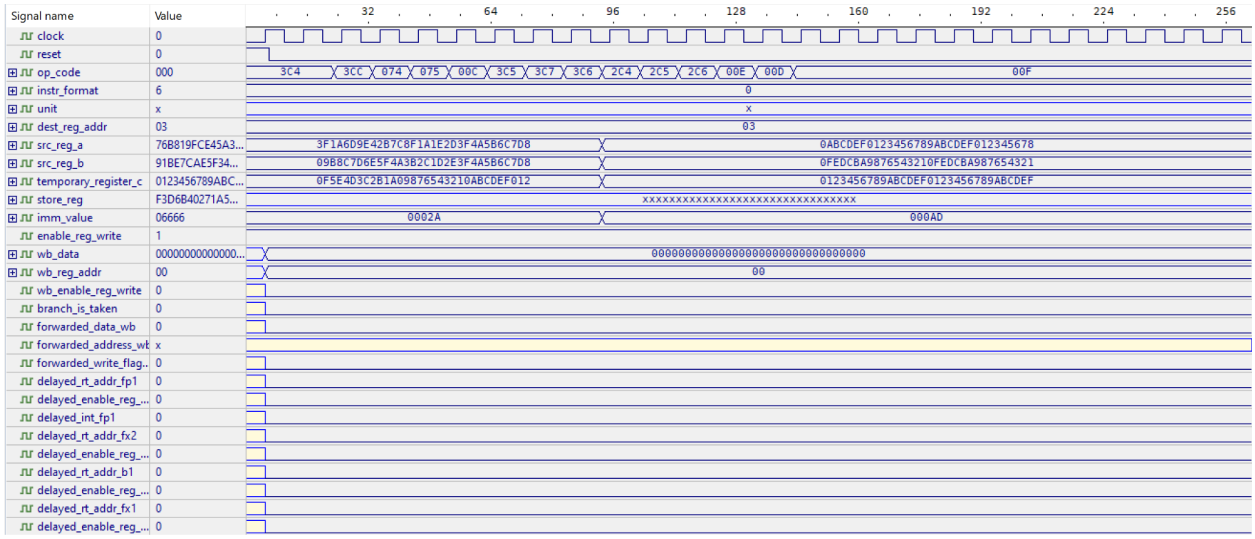
```
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
src_reg_b = 128'h849F36E1D4B5A2FC94DC931849F36E1D;
@(posedge clock);
// clgth rt, ra, rb Compare Logical Greater Than Halfword
#3;
op_code   = 11'b01011001000;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
src_reg_b = 128'h3C2FA791D68BFE92EDD6B7B3C2FA791D;
@(posedge clock);
// clgt rt, ra, rb Compare Logical Greater Than Word
#3;
op_code   = 11'b01011000000;
src_reg_a = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h91BE7CAE5F34D2A4DB1E38C91BE7CAE5;
@(posedge clock);
// ceqhi rt, ra, imm10 Compare Equal Halfword Immediate
#3;
op_code = 8'b01111101;
instr_format = 4;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
imm_value = 10'b1011111111;
@(posedge clock);
// ceqi rt, ra, imm10 Compare Equal Word Immediate
#3;
op_code = 8'b01111100;
instr_format = 4;
src_reg_a = 128'h849F36E1D4B5A2FC94DC931849F36E1D;
imm_value = 10'b1111111111;
@(posedge clock);
// cgthi rt, ra, imm10 Compare Greater Than Halfword Immediate
#3;
op_code = 8'b01001101;
instr_format = 4;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
imm_value = 10'b1111111111;
```

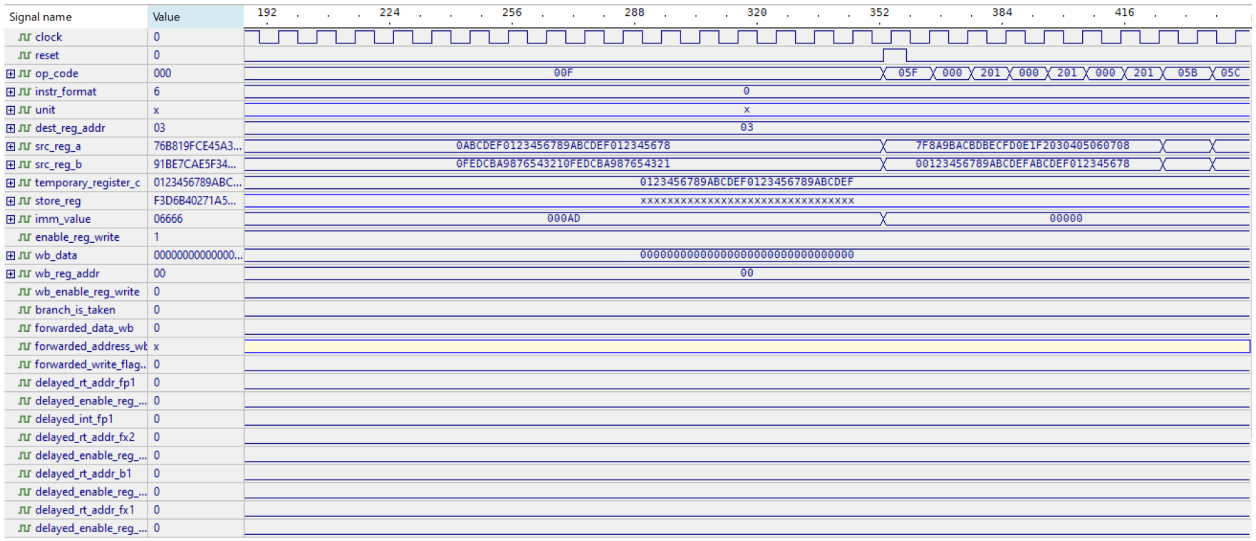
```
@(posedge clock);
// cgti rt, ra, imm10 Compare Greater Than Word Immediate
#3;
op_code = 8'b01001100;
instr_format = 4;
src_reg_a = 128'h3C2FA791D68BFE92EDD6B7B3C2FA791D;
imm_value = 10'b0111111111;
@(posedge clock);
// clgtbi rt, ra, imm10 Compare Logical Greater Than Byte Immediate
#3;
op_code = 8'b01011110;
instr_format = 4;
src_reg_a = 128'h57A8B910243E75FD124ED85957A8B910;
imm_value = 10'b1101111111;
@(posedge clock);
// clgthi rt, ra, imm10 Compare Logical Greater Than Halfword Immediate
#3;
op_code = 8'b01011101;
instr_format = 4;
src_reg_a = 128'h1D964C0EF27AB9C9A62ED4F91D964C0E;
imm_value = 10'b0111111111;
@(posedge clock);
// ilh rt, imm16 Immediate Load Halfword
#3;
op_code = 9'b010000011;
instr_format = 5;
src_reg_a = 128'hA3F1706B924E3D8B1C9F0B85A3F1706B;
imm_value = 16'b0110011001100110;
@(posedge clock);
// Immediate Load Halfword Upper
#3;
op_code = 9'b010000010;
instr_format = 5;
src_reg_a = 128'hF5E892A65B7C341F0E6F7DA6F5E892A6;
imm_value = 16'b1111111111111110;
```

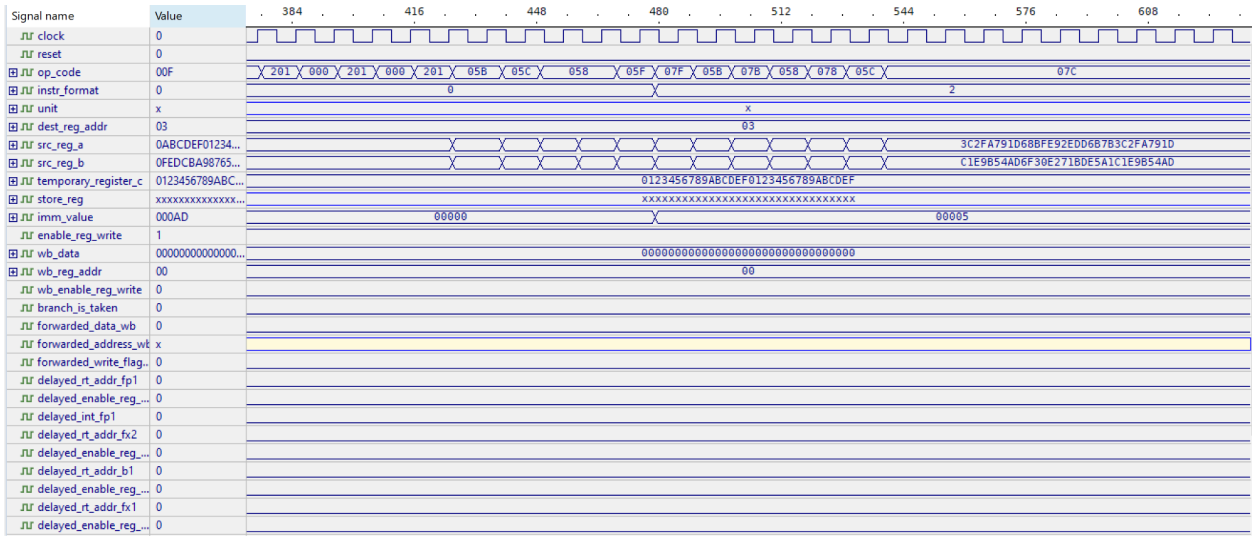
```
    @(posedge clock);
    // Immediate Load Word
    #3;
    op_code = 9'b010000001;
    instr_format = 5;
    src_reg_a = 128'hD8A21F3B5690C7E51D3BF48BD8A21F3B;
    imm_value = 32'b11111111111111111111111111111110;
    @(posedge clock);
    // iohl rt, imm16 Immediate Or Halfword Lower
    #3;
    op_code = 9'b011000001;
    instr_format = 5;
    src_reg_a = 128'h2E7F46C1583A9B04AD76A8C72E7F46C1;
    imm_value = 16'b0110011001100110;
    store_reg = 128'hBC30291F745ED6A58765DDE4BC30291F;
    @(posedge clock);
    // ila rt, imm18 Immediate Load Address
    #3;
    op_code = 7'b0100001;
    instr_format = 6;
    src_reg_a = 128'h76B819FCE45A3D76A5D49F9F76B819FC;
    imm_value = 18'b000110011001100110;
    store_reg = 128'hF3D6B40271A5C8E14FD38C1EF3D6B402;
    @(posedge clock);
    #3;
    op_code = 0;
    @(posedge clock);
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #100;
    op_code = 11'b00000000000;
    $stop;
  end
endmodule
```

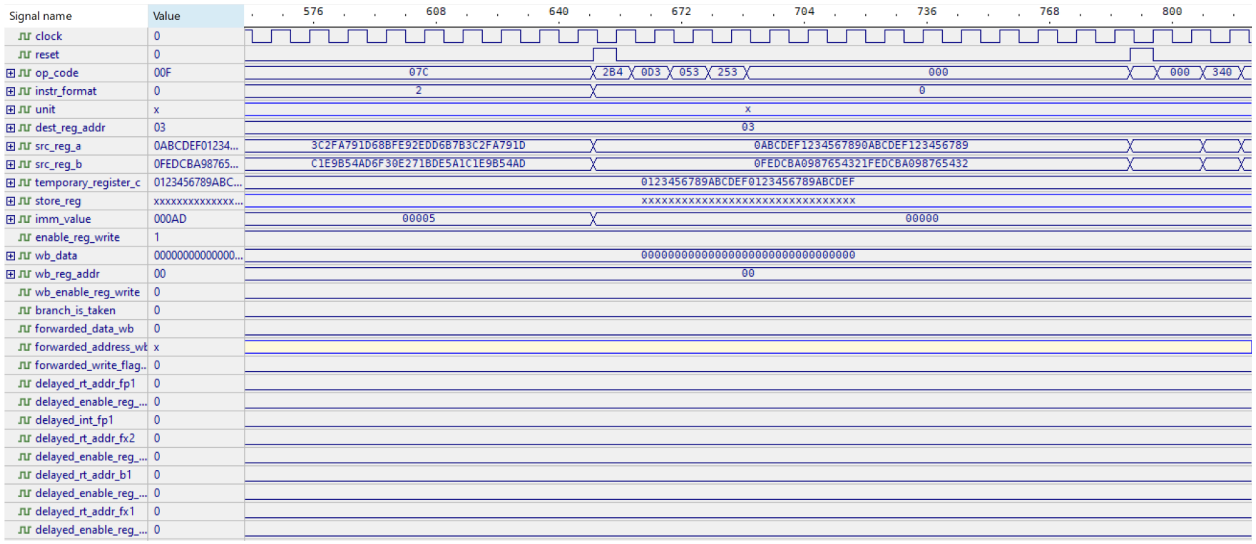
Even Pipe Waveform:

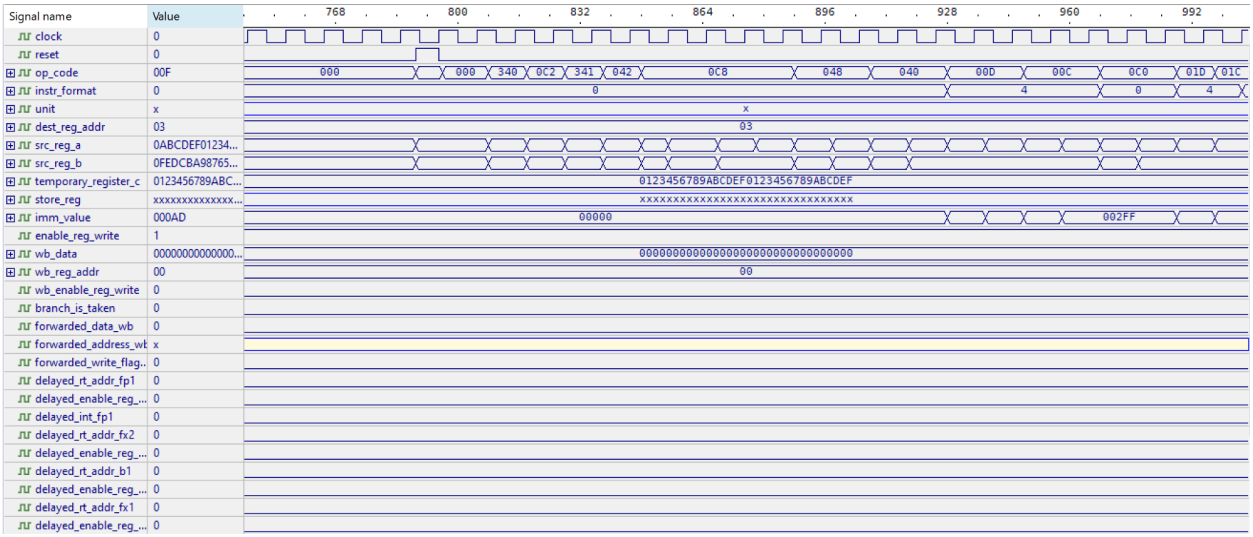


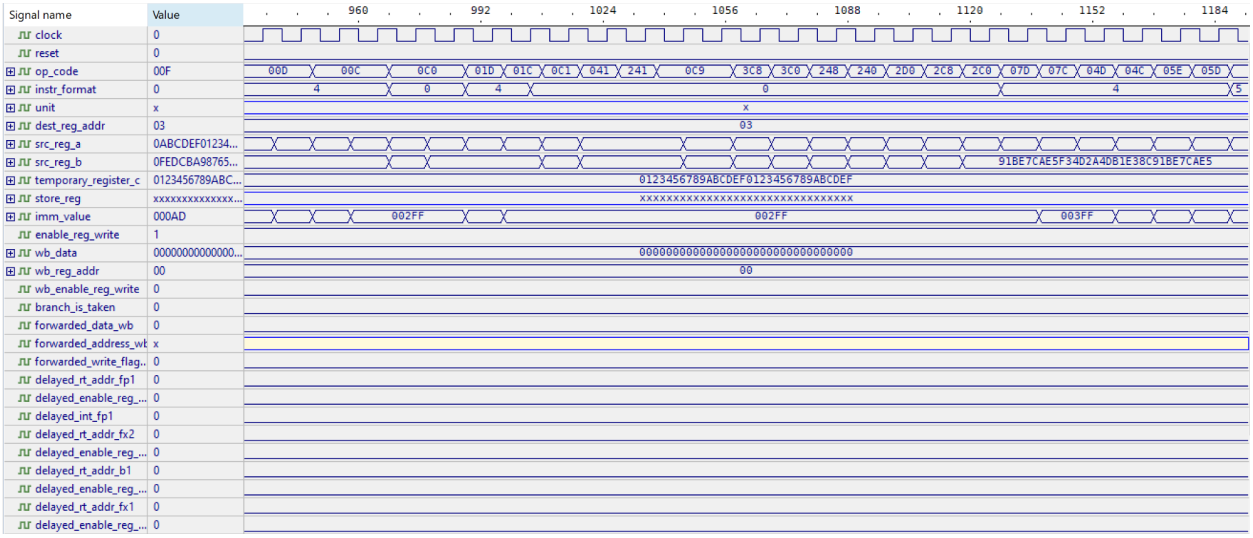


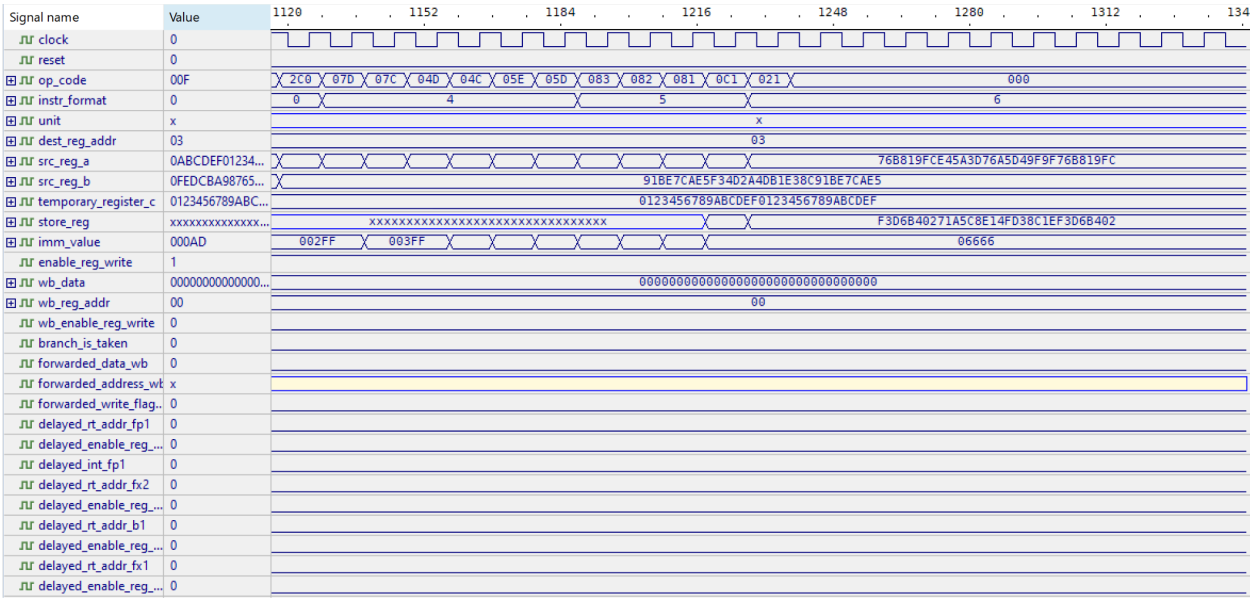

Signal name
Value
192 224 256 288 320 352 384 416
clock 0
reset 0
op_code 000 00F 05F 000 201 000 201 000 201 05B 05C
instr_format 6 0
unit x x
dest_reg_addr 03 03
src_reg_a 76B819FCE45A3... 0ABCDEF0123456789ABCDEF012345678 7F8A9BACBDBECFD0E1F2030405060708
src_reg_b 91BE7CAE5F34... 0FEDCBA9876543210FEDCBA987654321 00123456789ABCDEFABCDEF012345678
temporary_register_c 0123456789ABC... 0123456789ABCDEF0123456789ABCDEF
store_reg F3D6B40271A5... xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
imm_value 06666 000AD 00000
enable_reg_write 1
wb_data 00000000000000... 00000000000000000000000000000000
wb_reg_addr 00 00
wb_enable_reg_write 0
branch_is_taken 0
forwarded_data_wb 0
forwarded_address_wb x
forwarded_write_flag.. 0
delayed_rt_addr_fp1 0
delayed_enable_reg_... 0
delayed_int_fp1 0
delayed_rt_addr_fx2 0
delayed_enable_reg_... 0
delayed_rt_addr_b1 0
delayed_enable_reg_... 0
delayed_rt_addr_fx1 0
delayed_enable_reg_... 0


Signal name
Value
384 416 448 480 512 544 576 608
clock 0
reset 0
op_code 00F 201 000 201 000 201 05B 05C 058 05F 07F 05B 07B 058 078 05C 07C
instr_format 0 0 2
unit x x
dest_reg_addr 03 03
src_reg_a 0ABCDEF01234... 3C2FA791D68BFE92EDD6B7B3C2FA791D
src_reg_b 0FEDCBA98765... C1E9B54AD6F30E271BDE5A1C1E9B54AD
temporary_register_c 0123456789ABC... 0123456789ABCDEF0123456789ABCDEF
store_reg xxxxxxxxxxxxxx... xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
imm_value 000AD 00000 00005
enable_reg_write 1
wb_data 00000000000000... 00000000000000000000000000000000
wb_reg_addr 00 00
wb_enable_reg_write 0
branch_is_taken 0
forwarded_data_wb 0
forwarded_address_wb x
forwarded_write_flag.. 0
delayed_rt_addr_fp1 0
delayed_enable_reg_... 0
delayed_int_fp1 0
delayed_rt_addr_fx2 0
delayed_enable_reg_... 0
delayed_rt_addr_b1 0
delayed_enable_reg_... 0
delayed_rt_addr_fx1 0
delayed_enable_reg_... 0


Signal name
Value
576 608 640 672 704 736 768 800
clock 0
reset 0
op_code 00F 07C 284 0D3 053 253 000 000 340
instr_format 0 2 0
unit x x
dest_reg_addr 03 03
src_reg_a 0ABCDEF01234... 3C2FA791D68BFE92EDD6B7B3C2FA791D 0ABCDEF1234567890ABCDEF123456789
src_reg_b 0FEDCBA98765... C1E9B54AD6F30E271BDE5A1C1E9B54AD 0FEDCBA0987654321FEDCBA098765432
temporary_register_c 0123456789ABC... 0123456789ABCDEF0123456789ABCDEF
store_reg xxxxxxxxxxxxxx... xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
imm_value 000AD 00005 00000
enable_reg_write 1
wb_data 00000000000000... 00000000000000000000000000000000
wb_reg_addr 00 00
wb_enable_reg_write 0
branch_is_taken 0
forwarded_data_wb 0
forwarded_address_wb x
forwarded_write_flag.. 0
delayed_rt_addr_fp1 0
delayed_enable_reg_... 0
delayed_int_fp1 0
delayed_rt_addr_fx2 0
delayed_enable_reg_... 0
delayed_rt_addr_b1 0
delayed_enable_reg_... 0
delayed_rt_addr_fx1 0
delayed_enable_reg_... 0


Signal name
Value
768
800
832
864
896
928
960
992
clock
reset
op_code
instr_format
unit
dest_reg_addr
src_reg_a
src_reg_b
temporary_register_c
store_reg
imm_value
enable_reg_write
wb_data
wb_reg_addr
wb_enable_reg_write
branch_is_taken
forwarded_data_wb
forwarded_address_wb
forwarded_write_flag..
delayed_rt_addr_fp1
delayed_enable_reg_...
delayed_int_fp1
delayed_rt_addr_fx2
delayed_enable_reg_...
delayed_rt_addr_b1
delayed_enable_reg_...
delayed_rt_addr_fx1
delayed_enable_reg_...


Signal name
Value
960
992
1024
1056
1088
1120
1152
1184
clock
reset
op_code
instr_format
unit
dest_reg_addr
src_reg_a
src_reg_b
temporary_register_c
store_reg
imm_value
enable_reg_write
wb_data
wb_reg_addr
wb_enable_reg_write
branch_is_taken
forwarded_data_wb
forwarded_address_wb
forwarded_write_flag..
delayed_rt_addr_fp1
delayed_enable_reg_...
delayed_int_fp1
delayed_rt_addr_fx2
delayed_enable_reg_...
delayed_rt_addr_b1
delayed_enable_reg_...
delayed_rt_addr_fx1
delayed_enable_reg_...



Pipes:

Pipes:

The “Pipes” module plays an important role in orchestrating the various stages of the processor pipeline, managing everything from the instruction fetch to register file operations and even branch handling. This module serves as the backbone for ensuring the correctness of execution of instructions within the processor.

At the core, the module receives input signals such as clock signals, reset signals, instructions for both even and odd pipeline stages, program counters, opcode information, register addresses, immediate values, and flags indicating register writes. It will then process these inputs to manage pipeline stages, handle branch instructions, and facilitate register table operations.

The modules internal signals help with operations such as forwarding data between pipeline stages, managing write back operations to the register file, and even handling branch instructions. By coordinating these operations, the “Pipes” module ensures efficient instruction execution while also minimizing the potential errors such as data errors and control errors.

Overall, the “Pipes” module acts as the control center of the processor pipeline, enabling successful instruction execution and efficient utilization of hardware resources. Its design and functionality are essential for the overall performance and reliability of the processor architecture.

Pipes Code:

```
/*************************************************************************************************************
 * Module: Pipes
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *     This module orchestrates the pipeline stages of the processor, including the Register File, Instruction
 *     Fetch (IF), Even and Odd instruction decoding, forwarding logic, and branch handling.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - instruction_even: Instruction from the decoder for even pipeline stage
 *   - instruction_odd: Instruction from the decoder for odd pipeline stage
 *   - program_counter: Program counter from the IF stage
 *   - op_code_even, op_code_odd: op_code of the instruction for even and odd pipeline stages
 *   - format_is_even, format_is_odd: Format of the instruction for even and odd pipeline stages
 *   - unit_is_even, unit_is_odd: Destination unit of the instruction for even and odd pipeline stages
 *   - rt_address_even, rt_address_odd: Destination register addresses for even and odd pipeline stages
 *   - ra_even, rb_even, rc_even, ra_odd, rb_odd, rt_st_odd: Register values from the Register Table
 *   - immediate_even, immediate_odd: Immediate values for even and odd pipeline stages
 *   - register_write_even, register_write_odd: Flag indicating whether the current instruction writes to the register table
 *   - initial_odd: Flag indicating if the odd instruction is the initial_ in the pair
 *-------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - program_counter_wb: New program counter for branch instruction handling
 *   - branch_is_taken: Signal indicating if a branch was taken
 *-------------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - write_back_data_even, write_back_data_odd: Values to be written back to the register table for even and odd pipeline stages
 *   - rt_address_even_wb, rt_address_odd_wb: Destination register addresses for write-back for even and odd stages
 *   - register_write_even_wb, register_write_odd_wb: Flag indicating whether the write-back data will be written to the register table
 *   - forward_values_even, forward_values_odd: Pipe shift registers of values ready to be forwarded for even and odd pipeline stages
 *   - forward_address_even_wb, forward_address_odd_wb: Destination addresses of values to be forwarded for even and odd pipeline stages
 *   - forward_write_even_wb, forward_write_odd_wb: Flag indicating whether forwarded values will be written to the register table for even and
odd stages
```

```
 *   - disable_branch: Signal indicating if a branch is taken and the branch instruction is initial_ in the pair
 *   - format_is_even_live: Format of the instruction for the even stage, only valid if branch not taken by the initial_ instruction
 *   - op_code_even_live: op_code of the instruction for the even stage, only valid if branch not taken by the initial_ instruction
 *   - delay_rt_address_even, delay_rt_address_odd: Delayed destination register addresses for RAW Error handling
 *   - delay_register_write_even, delay_register_write_odd: Delayed flag indicating whether the instruction writes to the register table for RAW
Error handling
 *   - delay_rt_address_fp1, delay_register_write_fp1, delay_integer_fp1: Delayed signals for FP1 stage for RAW Error handling
 *   - delay_rt_address_fx2, delay_register_write_fx2: Delayed signals for FX2 stage for RAW Error handling
 *   - delay_rt_address_b1, delay_register_write_b1: Delayed signals for B1 stage for RAW Error handling
 *   - delay_rt_address_fx1, delay_register_write_fx1: Delayed signals for FX1 stage for RAW Error handling
 *   - delay_rt_address_p1, delay_register_write_p1: Delayed signals for P1 stage for RAW Error handling
 *   - delay_rt_address_ls1, delay_register_write_ls1: Delayed signals for LS1 stage for RAW Error handling
 ************************************************************************************************************/

module Pipes (
    clock,
    reset,
    instruction_even,
    instruction_odd,
    program_counter,
    program_counter_wb,
    branch_is_taken,
    op_code_even,
    op_code_odd,
    unit_is_even,
    unit_is_odd,
    rt_address_even,
    rt_address_odd,
    format_is_even,
    format_is_odd,
    immediate_even,
    immediate_odd,
    register_write_even,
    register_write_odd,
    initial_odd,
    delay_rt_address_even,
    delay_register_write_even,
    delay_rt_address_odd,
    delay_register_write_odd,
    delay_rt_address_fp1,
    delay_register_write_fp1,
    delay_integer_fp1,
    delay_rt_address_fx2,
    delay_register_write_fx2,
    delay_rt_address_b1,
    delay_register_write_b1,
    delay_rt_address_fx1,
    delay_register_write_fx1,
    delay_rt_address_p1,
    delay_register_write_p1,
    delay_rt_address_ls1,
    delay_register_write_ls1
);

  input logic clock, reset;

  // Instructions from the decoder
  input logic [0:31] instruction_even, instruction_odd;
  // Program counter from the Instruction Fetch stage
  input logic [7:0] program_counter;
```

```
// Nets from decode logic
// Format of instruction
input logic [2:0] format_is_even, format_is_odd;
// op_code of instruction (used with format)
input logic [0:10] op_code_even, op_code_odd;
// Destination unit of instruction; Order of: FP, FX2, Byte, FX1 (Even); Perm, LS, Br (Odd)
input logic [1:0] unit_is_even, unit_is_odd;
// Destination register addresses
input logic [0:6] rt_address_even, rt_address_odd;
// Register values from Register Table
logic [0:127] ra_even, rb_even, rc_even, ra_odd, rb_odd, rt_st_odd;
// Full possible immediate value (used with format)
input logic [0:17] immediate_even, immediate_odd;
// 1 if instruction will write to rt, else 0
input logic register_write_even, register_write_odd;
// 1 if odd instruction is initial_ in pair, 0 else; Used for branch flushing
input initial_odd;

// Signals for writing back to Register Table
// Values to be written back to Register Table
logic [0:127] write_back_data_even, write_back_data_odd;
// Destination register addresses
logic [0:6] rt_address_even_wb, rt_address_odd_wb;
// 1 if instruction will write to rt, else 0
logic register_write_even_wb, register_write_odd_wb;

// Pipe shift registers of values ready to be forwarded
logic [6:0][0:127] forward_values_even, forward_values_odd;
// Destinations of values to be forwarded
logic [6:0][0:6] forward_address_even_wb, forward_address_odd_wb;
// Will forwarded values be written to register?
logic [6:0] forward_write_even_wb, forward_write_odd_wb;
// Updated input values
logic [0:127]
   forward_even_ra,
   forward_even_rb,
   forward_even_rc,
   forward_odd_ra,
   forward_odd_rb,
   forward_odd_store_reg;

// New program counter for branch
output logic [7:0] program_counter_wb;
// Was branch taken?
output logic branch_is_taken;
// If branch is taken and branch instruction is initial_ in pair, kill twin instruction
logic disable_branch;
// Format of instruction, only valid if branch not taken by initial_ instruction
logic [2:0] format_is_even_live;
// op_code of instruction, only valid if branch not taken by initial_ instruction
logic [0:10] op_code_even_live;

// Destination register for rt_wb
output logic [0:6] delay_rt_address_even, delay_rt_address_odd;
// Will rt_wb write to Register Table
output logic delay_register_write_even, delay_register_write_odd;
// Destination register for rt_wb
output logic [6:0][0:6] delay_rt_address_fp1;
// Will rt_wb write to Register Table
output logic [6:0] delay_register_write_fp1;
// Will fp1 write an int result
```

```
output logic [6:0] delay_integer_fp1;
// Destination register for rt_wb
output logic [3:0][0:6] delay_rt_address_fx2;
// Will rt_wb write to Register Table
output logic [3:0] delay_register_write_fx2;
// Destination register for rt_wb
output logic [3:0][0:6] delay_rt_address_b1;
// Will rt_wb write to Register Table
output logic [3:0] delay_register_write_b1;
// Destination register for rt_wb
output logic [1:0][0:6] delay_rt_address_fx1;
// Will rt_wb write to Register Table
output logic [1:0] delay_register_write_fx1;
// Destination register for rt_wb
output logic [3:0][0:6] delay_rt_address_p1;
// Will rt_wb write to Register Table
output logic [3:0] delay_register_write_p1;
// Destination register for rt_wb
output logic [5:0][0:6] delay_rt_address_ls1;
// Will rt_wb write to Register Table
output logic [5:0] delay_register_write_ls1;

Register_File rf (
    .clock(clock),
    .reset(reset),
    .instruction_even(instruction_even),
    .instruction_odd(instruction_odd),
    .ra_even_input(ra_even),
    .rb_even_input(rb_even),
    .rc_even_input(rc_even),
    .ra_odd_input(ra_odd),
    .rb_odd_input(rb_odd),
    .rt_st_odd_input(rt_st_odd),
    .rt_address_even_input(rt_address_even_wb),
    .rt_address_odd_input(rt_address_odd_wb),
    .rt_even_input(write_back_data_even),
    .rt_odd_input(write_back_data_odd),
    .register_write_even_input(register_write_even_wb),
    .register_write_odd_input(register_write_odd_wb)
);
Even_Pipe ev (
    .clock(clock),
    .reset(reset),
    .op_code(op_code_even_live),
    .instr_format(format_is_even_live),
    .unit(unit_is_even),
    .dest_reg_addr(rt_address_even),
    .src_reg_a(forward_even_ra),
    .src_reg_b(forward_even_rb),
    .temporary_register_c(forward_even_rc),
    .imm_value(immediate_even),
    .enable_reg_write(register_write_even),
    .wb_data(write_back_data_even),
    .wb_reg_addr(rt_address_even_wb),
    .wb_enable_reg_write(register_write_even_wb),
    .branch_is_taken(branch_is_taken),
    .forwarded_data_wb(forward_values_even),
    .forwarded_address_wb(forward_address_even_wb),
    .forwarded_write_flag_wb(forward_write_even_wb),
    .delayed_rt_addr_fp1(delay_rt_address_fp1),
    .delayed_enable_reg_write_fp1(delay_register_write_fp1),
```

```
    .delayed_int_fp1(delay_integer_fp1),
    .delayed_rt_addr_fx2(delay_rt_address_fx2),
    .delayed_enable_reg_write_fx2(delay_register_write_fx2),
    .delayed_rt_addr_b1(delay_rt_address_b1),
    .delayed_enable_reg_write_b1(delay_register_write_b1),
    .delayed_rt_addr_fx1(delay_rt_address_fx1),
    .delayed_enable_reg_write_fx1(delay_register_write_fx1)
);
Odd_Pipe od (
    .clock(clock),
    .reset(reset),
    .op_code(op_code_odd),
    .instr_format(format_is_odd),
    .unit(unit_is_odd),
    .dest_reg_addr(rt_address_odd),
    .src_reg_a(forward_odd_ra),
    .src_reg_b(forward_odd_rb),
    .store_reg(forward_odd_store_reg),
    .imm_value(immediate_odd),
    .enable_reg_write(register_write_odd),
    .program_counter_input(program_counter),
    .wb_data(write_back_data_odd),
    .wb_reg_addr(rt_address_odd_wb),
    .wb_enable_reg_write(register_write_odd_wb),
    .program_counter_wb(program_counter_wb),
    .branch_is_taken(branch_is_taken),
    .forwarded_data_wb(forward_values_odd),
    .forwarded_address_wb(forward_address_odd_wb),
    .forwarded_write_flag_wb(forward_write_odd_wb),
    .initial_(initial_odd),
    .disable_branch(disable_branch),
    .delayed_ls1_register_address(delay_rt_address_p1),
    .delayed_ls1_register_write(delay_register_write_p1),
    .delayed_p1_register_address(delay_rt_address_ls1),
    .delayed_p1_register_write(delay_register_write_ls1)
);
Forward fwd (
    .clock(clock),
    .reset(reset),
    .instruction_even(instruction_even),
    .instruction_odd(instruction_odd),
    .ra_even_input(ra_even),
    .rb_even_input(rb_even),
    .rc_even_input(rc_even),
    .ra_odd_input(ra_odd),
    .rb_odd_input(rb_odd),
    .rt_st_odd_input(rt_st_odd),
    .ra_even_fwd_output(forward_even_ra),
    .rb_even_fwd_output(forward_even_rb),
    .rc_even_fwd_output(forward_even_rc),
    .ra_odd_fwd_output(forward_odd_ra),
    .rb_odd_fwd_output(forward_odd_rb),
    .rt_st_odd_fwd_output(forward_odd_store_reg),
    .fw_even_wb_input(forward_values_even),
    .fw_addr_even_wb_input(forward_address_even_wb),
    .fw_write_even_wb_input(forward_write_even_wb),
    .fw_odd_wb_input(forward_values_odd),
    .fw_addr_odd_wb_input(forward_address_odd_wb),
    .fw_write_odd_wb_input(forward_write_odd_wb)
);
```

```
  always_comb begin
    // Check if the branch instruction is taken and if it is the initial_ instruction in the pair
    if (disable_branch == 0) begin
      format_is_even_live = format_is_even;
      op_code_even_live   = op_code_even;
    end else begin
      format_is_even_live = 0;
      op_code_even_live   = 0;
    end
    delay_rt_address_even = rt_address_even;
    delay_register_write_even = register_write_even;
    delay_rt_address_odd = rt_address_odd;
    delay_register_write_odd = register_write_odd;
  end
endmodule
```

Pipes Test Bench:

“Pipes” testbench module serves as another important component in ensuring the proper functionality and performance of the main “Pipes” module. This testbench generates a simulation for the “Pipes” module and monitors its output to verify successful execution. By simulating the various scenarios and edge cases, this testbench helps to validate the behavior of the “Pipes” module under unique conditions.

The inputs to the testbench include clock and reset signals, instructions for both even and odd pipeline stages, program counters, opcode information, register addresses, immediate values, and flags indicating register writes. These inputs are manipulated in a way to create a diverse test case, covering a wide range of potential scenarios that the processor may encounter in real world operations.

During simulation, the testbench toggles the clock signal to a regular interval to simulate the oscillation of a real clock. It then sets unique values for instructions, opcodes, and other control signals to simulate the execution of various instructions within the processor's pipeline. By understanding and looking at the values in the output, such as the program counter, branch taken signals, and delayed signals for error handling, the testbench can verify the successful behavior of the “Pipes” module and detect any errors that may occur. Overall, the “Pipes” testbench module plays an important role in helping the reliability of the processor design by thoroughly testing the functionality under broad conditions.

Pipes Test Bench Code:

```
/*****************************************************************************************************
 * Module: Pipes Testbench
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------
 * Description:
```

```
*     This module serves as the testbench for the Pipes module. It generates stimulus for the Pipes module
*     and monitors its outputs for correctness.
*-----------------------------------------------------------------------------------------------------
* Inputs:
*   - clock: Clock signal
*   - reset: Reset signal
*   - instruction_even: Instruction from the decoder for even pipeline stage
*   - instruction_odd: Instruction from the decoder for odd pipeline stage
*   - program_counter: Program counter from the IF stage
*   - op_code_even, op_code_odd: op_code of the instruction for even and odd pipeline stages
*   - format_is_even, format_is_odd: Format of the instruction for even and odd pipeline stages
*   - unit_is_even, unit_is_odd: Destination unit of the instruction for even and odd pipeline stages
*   - rt_address_even, rt_address_odd: Destination register addresses for even and odd pipeline stages
*   - ra_even, rb_even, rc_even, ra_odd, rb_odd, rt_st_odd: Register values from the Register Table
*   - immediate_even, immediate_odd: Immediate values for even and odd pipeline stages
*   - register_write_even, register_write_odd: Flag indicating whether the current instruction writes to the register table
*   - initial_odd: Flag indicating if the odd instruction is the first in the pair
*-----------------------------------------------------------------------------------------------------
* Outputs:
*   - program_counter_wb: New program counter for branch instruction handling
*   - branch_is_taken: Signal indicating if a branch was taken
*-----------------------------------------------------------------------------------------------------
* Internal Signals:
*   - rt_even_wb, rt_odd_wb: Values to be written back to the register table for even and odd pipeline stages
*   - rt_address_even_wb, rt_address_odd_wb: Destination register addresses for write-back for even and odd stages
*   - register_write_even_wb, register_write_odd_wb: Flag indicating whether the write-back data will be written to the register table
*   - fw_even_wb, fw_odd_wb: Pipe shift registers of values ready to be forwarded for even and odd pipeline stages
*   - fw_addr_even_wb, fw_addr_odd_wb: Destination addresses of values to be forwarded for even and odd pipeline stages
*   - fw_write_even_wb, fw_write_odd_wb: Flag indicating whether forwarded values will be written to the register table for even and odd stages
*   - branch_kill: Signal indicating if a branch is taken and the branch instruction is first in the pair
*   - format_is_even_live: Format of the instruction for the even stage, only valid if branch not taken by the first instruction
*   - op_code_even_live: op_code of the instruction for the even stage, only valid if branch not taken by the first instruction
*   - delay_rt_address_even, delay_rt_address_odd: Delayed destination register addresses for RAW hazard handling
*   - delay_register_write_even, delay_register_write_odd: Delayed flag indicating whether the instruction writes to the register table for RAW
hazard handling
```

```
*   - delay_rt_address_fp1, delay_register_write_fp1, delay_integer_fp1: Delayed signals for FP1 stage for RAW hazard handling
*   - delay_rt_address_fx2, delay_register_write_fx2: Delayed signals for FX2 stage for RAW hazard handling
*   - delay_rt_address_b1, delay_register_write_b1: Delayed signals for B1 stage for RAW hazard handling
*   - delay_rt_address_fx1, delay_register_write_fx1: Delayed signals for FX1 stage for RAW hazard handling
*   - delay_rt_address_p1, delay_register_write_p1: Delayed signals for P1 stage for RAW hazard handling
*   - delay_rt_address_ls1, delay_register_write_ls1: Delayed signals for LS1 stage for RAW hazard handling
**********************************************************************************************************/

module Pipes_TB ();
  logic clock, reset;

  // Instructions from the decoder
  logic [0:31] instruction_even, instruction_odd;
  // Program counter from the Instruction Fetch stage
  logic [7:0] program_counter;

  // Signals for handling branches
  // New program counter for branch
  logic [7:0] program_counter_wb;
  // Was branch taken?
  logic branch_is_taken;
  // Signal indicating the opcode for the even pipeline
  logic op_code_even;
  // Signal indicating the opcode for the odd pipeline
  logic op_code_odd;
  // Signal indicating whether the execution unit for the even pipeline is active
  logic unit_is_even;
  // Signal indicating whether the execution unit for the odd pipeline is active
  logic unit_is_odd;
  // Signal representing the register address for the even pipeline
  logic rt_address_even;
  // Signal representing the register address for the odd pipeline
  logic rt_address_odd;
  // Signal indicating the instruction format for the even pipeline
  logic format_is_even;
```

```
// Signal indicating the instruction format for the odd pipeline
logic format_is_odd;
// Signal representing the immediate value for the even pipeline
logic immediate_even;
// Signal representing the immediate value for the odd pipeline
logic immediate_odd;
// Signal indicating whether register write is enabled for the even pipeline
logic register_write_even;
// Signal indicating whether register write is enabled for the odd pipeline
logic register_write_odd;
// Signal indicating the initial state for the odd pipeline
logic initial_odd;
// Signal representing the delayed register address for the even pipeline
logic delay_rt_address_even;
// Signal representing the delayed register write for the even pipeline
logic delay_register_write_even;
// Signal representing the delayed register address for the odd pipeline
logic delay_rt_address_odd;
// Signal representing the delayed register write for the odd pipeline
logic delay_register_write_odd;
// Signal representing the delayed register address for the FP1 unit
logic delay_rt_address_fp1;
// Signal representing the delayed register write for the FP1 unit
logic delay_register_write_fp1;
// Signal representing the delayed integer for the FP1 unit
logic delay_integer_fp1;
// Signal representing the delayed register address for the FX2 unit
logic delay_rt_address_fx2;
// Signal representing the delayed register write for the FX2 unit
logic delay_register_write_fx2;
// Signal representing the delayed register address for the B1 unit
logic delay_rt_address_b1;
// Signal representing the delayed register write for the B1 unit
logic delay_register_write_b1;
// Signal representing the delayed register address for the FX1 unit
```

```
logic delay_rt_address_fx1;
// Signal representing the delayed register write for the FX1 unit
logic delay_register_write_fx1;
// Signal representing the delayed register address for the P1 unit
logic delay_rt_address_p1;
// Signal representing the delayed register write for the P1 unit
logic delay_register_write_p1;
// Signal representing the delayed register address for the LS1 unit
logic delay_rt_address_ls1;
// Signal representing the delayed register write for the LS1 unit
logic delay_register_write_ls1;

Pipes dut (
    clock,
    reset,
    instruction_even,
    instruction_odd,
    program_counter,
    program_counter_wb,
    branch_is_taken,
    op_code_even,
    op_code_odd,
    unit_is_even,
    unit_is_odd,
    rt_address_even,
    rt_address_odd,
    format_is_even,
    format_is_odd,
    immediate_even,
    immediate_odd,
    register_write_even,
    register_write_odd,
    initial_odd,
    delay_rt_address_even,
    delay_register_write_even,
```

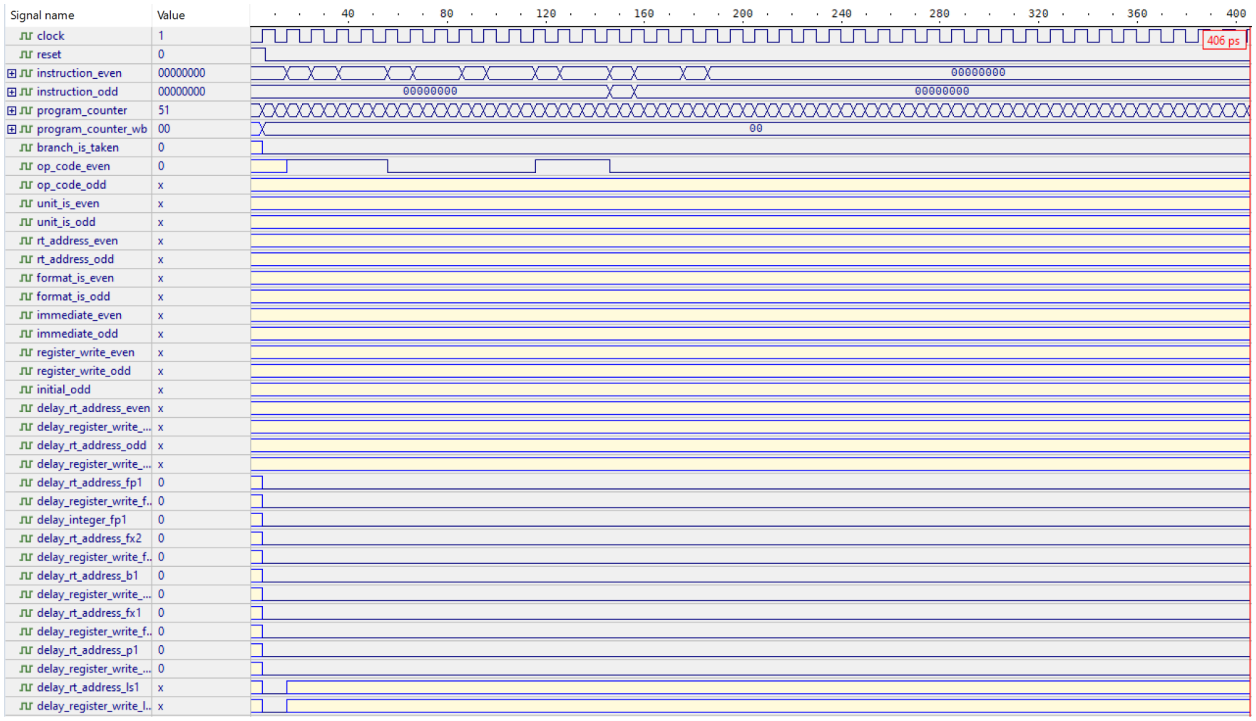
```
    delay_rt_address_odd,
    delay_register_write_odd,
    delay_rt_address_fp1,
    delay_register_write_fp1,
    delay_integer_fp1,
    delay_rt_address_fx2,
    delay_register_write_fx2,
    delay_rt_address_b1,
    delay_register_write_b1,
    delay_rt_address_fx1,
    delay_register_write_fx1,
    delay_rt_address_p1,
    delay_register_write_p1,
    delay_rt_address_ls1,
    delay_register_write_ls1
);

// Set the initial state of the clock to zero
initial clock = 0;

// Toggle the clock value every 5 time units to simulate oscillation
always begin
  #5 clock = ~clock;
  program_counter = program_counter + 1;
end

initial begin
  reset = 1;
  program_counter = 0;
  instruction_even = 0;
  instruction_odd = 0;
  #6;
  // At 11ns, disable the reset, enabling the unit
  reset = 0;
  // Wait for the positive edge of the clock, then pause the simulation for 1 time unit; This occurs at 15ns
```

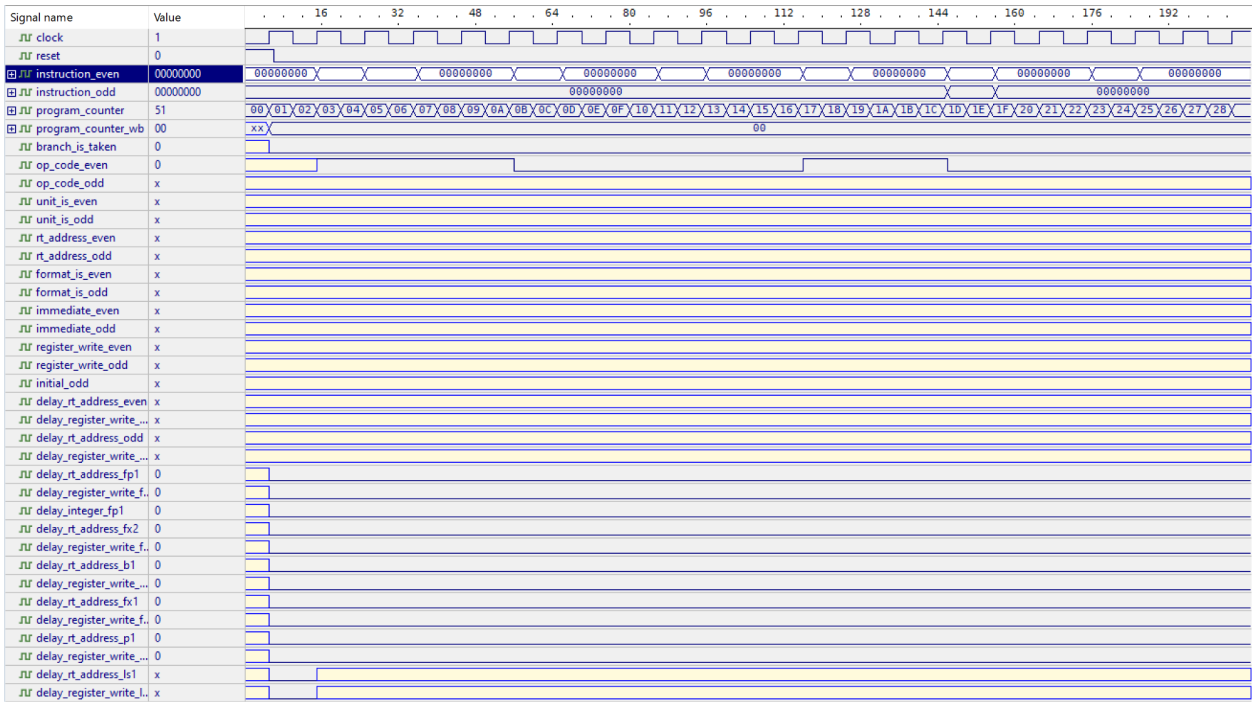
```
@(posedge clock);
// Set the op_code for the "Immediate Load Halfword" operation (ilh)
op_code_even = 9'b010000011;
instruction_even = 32'b10101111111001100101011100010010 0;
// Set the op_code for the "No Operation" operation (nop)
instruction_odd = 32'b0;
// Wait for the positive edge of the clock, then pause the simulation for 1 time unit; This occurs at 15ns
@(posedge clock);
// Set the op_code for the "Immediate Load Halfword" operation (ilh)
op_code_even = 9'b010000011;
instruction_even = 32'b11011011000100110111101100001011;
// Set the op_code for the "No Operation" operation (nop)
instruction_odd = 32'b0;
@(posedge clock);
#1;
instruction_even = 0;
instruction_odd  = 0;
@(posedge clock);
@(posedge clock);
#1;
// Set the op_code for the "Add Halfword" operation (ah)
op_code_even = 11'b00011001000;
instruction_even = 32'b01010101001111100000110011101000;
// Set the op_code for the "No Operation" operation (nop)
instruction_odd = 32'b0;
@(posedge clock);
#1;
instruction_even = 0;
instruction_odd  = 0;
@(posedge clock);
@(posedge clock);
#1;
// Set the op_code for the "Immediate Load Halfword" operation (ilh)
instruction_even = 32'b11101010100001010111101001101110;
// Set the op_code for the "No Operation" operation (nop)
```

```
instruction_odd  = 32'b0;
@(posedge clock);
#1;
instruction_even = 0;
instruction_odd  = 0;
@(posedge clock);
@(posedge clock);
#1;
// Set the op_code for the "Shift Left Halfword Immediate" operation (shlhi)
op_code_even = 11'b00001111111;
instruction_even = 32'b00110010111011100001101101011110;
instruction_odd = 0;
@(posedge clock);
#1;
instruction_even = 0;
instruction_odd  = 0;
@(posedge clock);
// Wait for the positive edge of the clock, then pause the simulation for 1 time unit; This occurs at 35ns
@(posedge clock);
#1;
// Set the opcode for the "Rotate Halfword" operation (roth)
op_code_even = 11'b00001011100;
instruction_even = 32'b10011100100100100110101111000101;
// Set the opcode for the "Rotate Halfword Immediate" operation (rothi)
op_code_even = 11'b00001111100;
instruction_odd = 32'b01101001111010100110010001111011;
@(posedge clock);
#1;
instruction_even = 0;
instruction_odd  = 0;
@(posedge clock);
// Wait for the positive edge of the clock, then pause the simulation for 1 time unit; This occurs at 35ns
@(posedge clock);
#1;
// Set the op_code for the "Rotate Halfword Immediate" operation (rotmah)
```

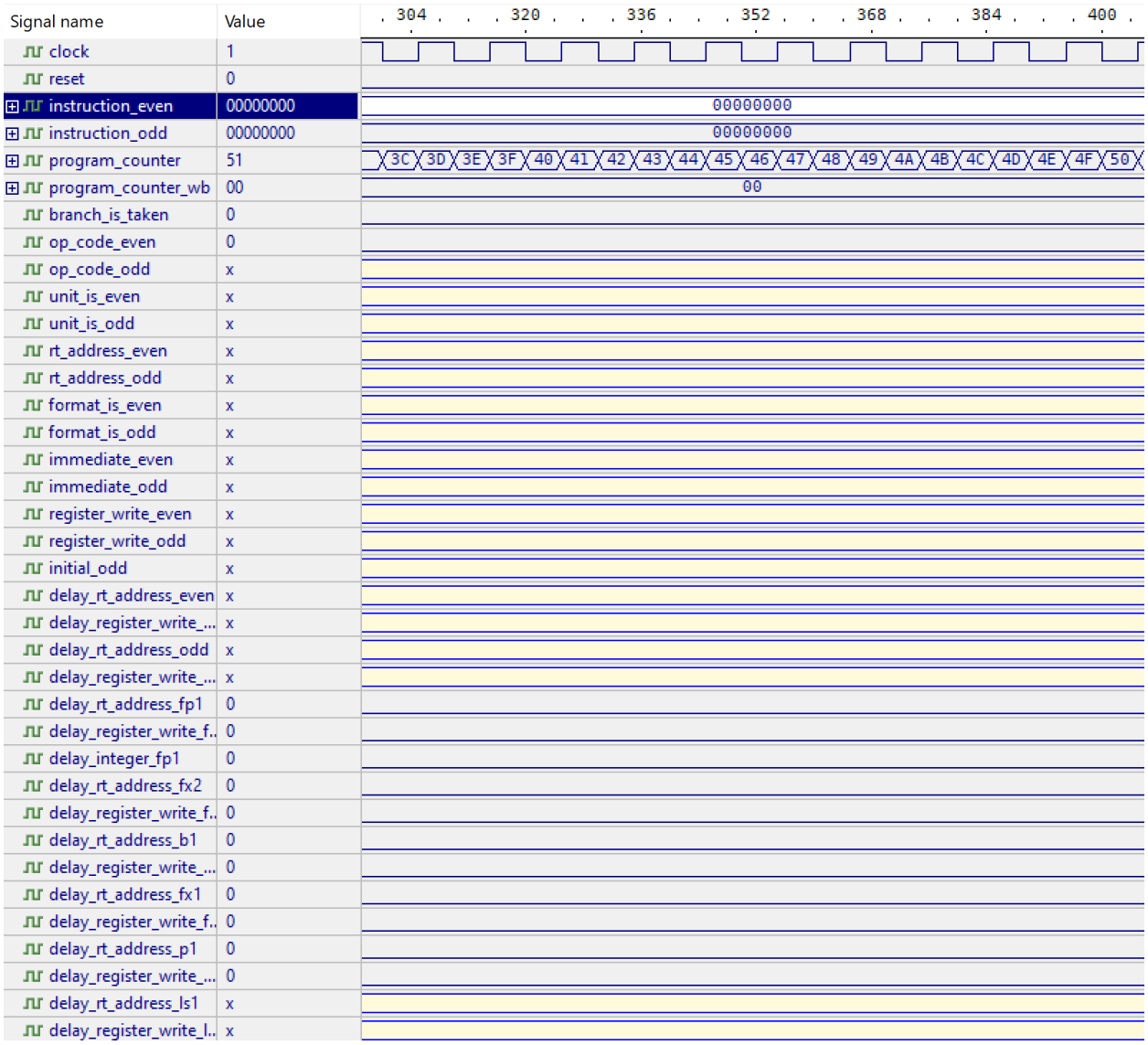
```
    op_code_even = 11'b00001111100;
    instruction_even = 32'b11010011111000100101101010100011;
    instruction_odd = 0;
    // Wait for the positive edge of the clock, then pause the simulation for 1 time unit; This occurs at 40ns
    @(posedge clock);
    #1;
    instruction_even = 0;
    instruction_odd  = 0;
    // Wait for the positive edge of the clock, then pause the simulation for 1 time unit; This occurs at 45ns
    @(posedge clock);
    #1;
    // Wait for the positive edge of the clock, then pause the simulation for 1 time unit; This occurs at 50ns
    @(posedge clock);
    #1;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #200;
    $stop;
  end
endmodule
```

Pipes Waveform:






Signal name
Value
160
176
192
208
224
240
256
272
288
304
320
336
352
clock 1
reset 0
instruction_even 00000000
instruction_odd 00000000
program_counter 51
program_counter_wb 00
branch_is_taken 0
op_code_even 0
op_code_odd x
unit_is_even x
unit_is_odd x
rt_address_even x
rt_address_odd x
format_is_even x
format_is_odd x
immediate_even x
immediate_odd x
register_write_even x
register_write_odd x
initial_odd x
delay_rt_address_even x
delay_register_write_... x
delay_rt_address_odd x
delay_register_write_... x
delay_rt_address_fp1 0
delay_register_write_f.. 0
delay_integer_fp1 0
delay_rt_address_fx2 0
delay_register_write_f.. 0
delay_rt_address_b1 0
delay_register_write_... 0
delay_rt_address_fx1 0
delay_register_write_f.. 0
delay_rt_address_p1 0
delay_register_write_... 0
delay_rt_address_ls1 x
delay_register_write_l.. x
00000000
00000000
00000000
1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46
00



Register:

Register:

The “Register File” module serves as a component that manages register file operations and implements forwarding logic within a processor. This modules importance is to ensure the efficient execution of instructions by accessing register values and data forwarding when needed.

At the core, the module takes inputs like clock and reset signals, instructions from both even and odd pipeline stages, register addresses, and values to be written to registers. It will then perform operations to read and update the register values based upon the incoming instructions and control signals.

The module's functionality can be divided into two main parts: combinational logic and sequential logic. In the combinational logic part, the module will read source register values based on the instructions received. Additionally, it also implements forwarding logic to ensure that the correct data is forwarded to correct pipeline stages if an instruction writes to the register table. In the sequential logic part, the modules will update the register file based on the incoming instructions, which will help the register values to be updated correctly on each clock cycle.

Overall, the “Register File” module is an important processor operation, using efficient management of the register values and helping to ensure successful execution of the instructions through data forwarding. Its implementation helps with the correct handling of register operations which overall contributes to the performance of the processor.

Register Code:

```
/*************************************************************************************************************
 * Module: Register File
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *                          This module handles register file operations including forwarding logic in a processor.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - instruction_even: Instruction from even pipeline stage
 *   - instruction_odd: Instruction from odd pipeline stage
 *   - ra_even_input: Value of register A for even pipeline stage
 *   - rb_even_input: Value of register B for even pipeline stage
 *   - rc_even_input: Value of register C for even pipeline stage
 *   - ra_odd_input: Value of register A for odd pipeline stage
 *   - rb_odd_input: Value of register B for odd pipeline stage
 *   - rt_st_odd_input: Value of temporary register rt_st for odd pipeline stage
 *   - rt_address_even_input: Destination register address to write to for even pipeline stage
```

```
*   - rt_address_odd_input: Destination register address to write to for odd pipeline stage
*   - rt_even_input: Value to write to the destination register for the even pipeline stage
*   - rt_odd_input: Value to write to the destination register for the odd pipeline stage
*   - register_write_even_input: Flag indicating whether the even instruction will write to a register
*   - register_write_odd_input: Flag indicating whether the odd instruction will write to a register
*-----------------------------------------------------------------------------------------------------
* Outputs:
*   - ra_even_input: Value of register A for the even pipeline stage
*   - rb_even_input: Value of register B for the even pipeline stage
*   - rc_even_input: Value of register C for the even pipeline stage
*   - ra_odd_input: Value of register A for the odd pipeline stage
*   - rb_odd_input: Value of register B for the odd pipeline stage
*   - rt_st_odd_input: Value of temporary register rt_st for the odd pipeline stage
*-----------------------------------------------------------------------------------------------------
* Internal Signals:
*   - registers: Register file storing register values
*   - i: 8-bit counter for the reset loop
*****************************************************************************************************/

module Register_File (
    clock,
    reset,
    instruction_even,
    instruction_odd,
    ra_even_input,
    rb_even_input,
    rc_even_input,
    ra_odd_input,
    rb_odd_input,
    rt_st_odd_input,
    rt_address_even_input,
    rt_address_odd_input,
    rt_even_input,
    rt_odd_input,
    register_write_even_input,
```

```
  register_write_odd_input
);

 input clock, reset;

 //Register File/Forwarding Stage
 // Input instructions for both even and odd cycles, received from the decoder
 input [0:31] instruction_even, instruction_odd;
 // Input values for 'ra', 'rb', 'rc' for both even and odd instructions, and 'rt_st' for odd instruction, retrieved from the Register Table
 output logic [0:127] ra_even_input, rb_even_input, rc_even_input, ra_odd_input, rb_odd_input, rt_st_odd_input;

 //Write Back Stage
 // Destination registers for even and odd instructions
 input [0:6] rt_address_even_input, rt_address_odd_input;
 // Values to be written to destination registers for even and odd instructions
 input [0:127] rt_even_input, rt_odd_input;
 // Flags indicating if the even and odd instructions will write to the register table
 input register_write_even_input, register_write_odd_input;

 //Internal Signals
 // Register File storing 128 128-bit registers
 logic [0:127] registers[0:127];
 // 8-bit counter used in reset loop
 logic [  7:0] i;

 always_comb begin
  // Read source register addresses from instructions
  rc_even_input = registers[instruction_even[25:31]];
  ra_even_input = registers[instruction_even[18:24]];
  rb_even_input = registers[instruction_even[11:17]];
  ra_odd_input = registers[instruction_odd[18:24]];
  rb_odd_input = registers[instruction_odd[11:17]];
  rt_st_odd_input = registers[instruction_odd[25:31]];

  // Forward data from even instruction if writing to register table
```

```
    if (register_write_even_input) begin
      if (instruction_even[25:31] == rt_address_even_input) rc_even_input = rt_even_input;
      if (instruction_even[18:24] == rt_address_even_input) ra_even_input = rt_even_input;
      if (instruction_even[11:17] == rt_address_even_input) rb_even_input = rt_even_input;
      if (instruction_odd[25:31] == rt_address_even_input) rt_st_odd_input = rt_even_input;
      if (instruction_odd[18:24] == rt_address_even_input) ra_odd_input = rt_even_input;
      if (instruction_odd[11:17] == rt_address_even_input) rb_odd_input = rt_even_input;
    end

    // Forward data from odd instruction if writing to register table
    if (register_write_odd_input) begin
      if (instruction_even[25:31] == rt_address_odd_input) rc_even_input = rt_odd_input;
      if (instruction_even[18:24] == rt_address_odd_input) ra_even_input = rt_odd_input;
      if (instruction_even[11:17] == rt_address_odd_input) rb_even_input = rt_odd_input;
      if (instruction_odd[25:31] == rt_address_odd_input) rt_st_odd_input = rt_odd_input;
      if (instruction_odd[18:24] == rt_address_odd_input) ra_odd_input = rt_odd_input;
      if (instruction_odd[11:17] == rt_address_odd_input) rb_odd_input = rt_odd_input;
    end
  end

  /* Sequential logic for updating the register file. On a reset, all registers are cleared. On each clock cycle, if the even or odd instruction
is writing to the register table, the corresponding register is updated with the new value. */
  always_ff @(posedge clock) begin
    if (reset) begin
      registers[127] <= 0;
      for (i = 0; i < 127; i = i + 1) registers[i] <= 0;
    end else begin
      if (register_write_even_input) registers[rt_address_even_input] <= rt_even_input;
      if (register_write_odd_input) registers[rt_address_odd_input] <= rt_odd_input;
    end
  end
endmodule
```

Register File Test Bench:

The “Register File” testbench module serves as a verification tool for the functionality of the Register File module. The Register File module is responsible for managing register file operations and forwarding logic within the processor.

In this testbench, a bunch of inputs are provided to simulate different scenarios and test the behavior of the Register File module under unique conditions. These inputs include clock signals, reset signals, instructions from both even and odd pipeline stages, register addresses, values to be written to registers, and flags indicating whether instructions will be written to the register table.

The testbench runs a series of simulations to evaluate how the Register File module handles register operations and forwarding logic. It will check whether the module correctly reads and updates the register values based on the incoming instructions and control signal. Also, it verifies the modules performance ability for data forwarding when needed, helping to have the correct data forwarded to subsequent pipeline stages. Through these simulations, the testbench helps to address the reliability and efficiency of the Register File module, contributing to the validation process and the processors overall functionality.

Register File Test Bench Code:

```
/*************************************************************************************************************
 * Module: Register File Testbench
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *                          This testbench module verifies the functionality of the Register File module, which handles register
 *     file operations including forwarding logic in a processor.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - instruction_even: Instruction from even pipeline stage
 *   - instruction_odd: Instruction from odd pipeline stage
 *   - ra_even_input: Value of register A for even pipeline stage
 *   - rb_even_input: Value of register B for even pipeline stage
 *   - rc_even_input: Value of register C for even pipeline stage
```

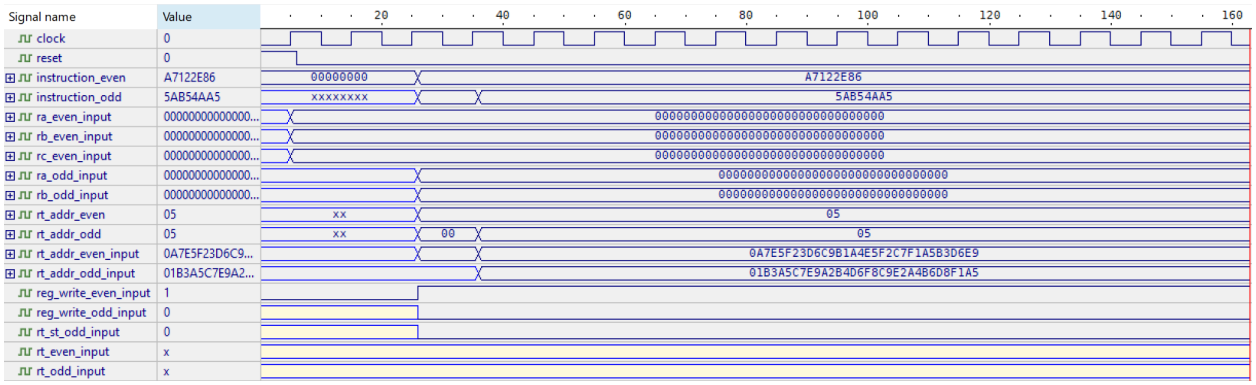
```
*   - ra_odd_input: Value of register A for odd pipeline stage
*   - rb_odd_input: Value of register B for odd pipeline stage
*   - rt_st_odd_input: Value of temporary register rt_st for odd pipeline stage
*   - rt_addr_even_input: Destination register address to write to for even pipeline stage
*   - rt_addr_odd_input: Destination register address to write to for odd pipeline stage
*   - rt_even_input: Value to write to the destination register for the even pipeline stage
*   - rt_odd_input: Value to write to the destination register for the odd pipeline stage
*   - reg_write_even_input: Flag indicating whether the even instruction will write to a register
*   - reg_write_odd_input: Flag indicating whether the odd instruction will write to a register
*-----------------------------------------------------------------------------------------------
* Outputs:
*   - ra_even_input: Value of register A for the even pipeline stage
*   - rb_even_input: Value of register B for the even pipeline stage
*   - rc_even_input: Value of register C for the even pipeline stage
*   - ra_odd_input: Value of register A for the odd pipeline stage
*   - rb_odd_input: Value of register B for the odd pipeline stage
*   - rt_st_odd_input: Value of temporary register rt_st for the odd pipeline stage
*-----------------------------------------------------------------------------------------------
* Internal Signals:
*   - registers: Register file storing register values
*   - i: 8-bit counter for the reset loop
************************************************************************************************/


module Register_File_TB ();
  logic clock, reset;


  // Input instructions for both even and odd cycles, received from the decoder
  logic [0:31] instruction_even, instruction_odd;
  // Input values for 'ra', 'rb', 'rc' for both even and odd instructions, retrieved from the Register Table
  logic [0:127] ra_even_input, rb_even_input, rc_even_input, ra_odd_input, rb_odd_input;
  // Destination registers for even and odd instructions
  logic [0:6] rt_addr_even, rt_addr_odd;
  // Values to be written to destination registers for even and odd instructions
  logic [0:127] rt_addr_even_input, rt_addr_odd_input;
  // Flags indicating if the even and odd instructions will write to the register table
```

```
logic reg_write_even_input, reg_write_odd_input;
// Indicates whether the data from the Store will be written to the odd pipe register file input
logic rt_st_odd_input;
// Indicates whether the data from the even pipe register file will be written to the odd pipe register file input
logic rt_even_input;
// Indicates whether the data from the odd pipe register file will be written to the odd pipe register file input
logic rt_odd_input;

Register_File dut (
    clock,
    reset,
    instruction_even,
    instruction_odd,
    ra_even_input,
    rb_even_input,
    rc_even_input,
    ra_odd_input,
    rb_odd_input,
    rt_st_odd_input,
    rt_addr_even_input,
    rt_addr_odd_input,
    rt_even_input,
    rt_odd_input,
    reg_write_even_input,
    reg_write_odd_input
);

initial clock = 0;

always begin
  #5 clock = ~clock;
end

initial begin
  reset = 1;
```

```
reg_write_even_input = 0;
instruction_even = 32'h00000000;
#6;
// Enable unit at 11ns
reset = 0;
@(posedge clock);
#1;
@(posedge clock);
#1;
reg_write_even_input = 1;
// shlh instruction with rb=3, ra=4, rt=5
instruction_even = 32'b10100111000100100010111010000110;
rt_addr_even = 7'b000101;
rt_addr_even_input = 128'h3F8A1B9E2C7F1A5B3D6E9C2F5A8B1C4;
// lnop instruction
instruction_odd = 32'b10101011010101101010101010101010;
reg_write_odd_input = 0;
rt_addr_odd = 7'b0000000;
// Writing the value rt_addr_even_input to address 7
@(posedge clock) #1;
// Attempting to read the value at address 7 for odd pipe while even pipe is also writing back to the same
// register file (7). Odd pipe is able to read the value from reg 7 before even pipe writes back a new value to it.
reg_write_even_input = 1;
// shlh instruction with rb=3, ra=4, rt=5
instruction_even = 32'b10100111000100100010111010000110;
// shlqi instruction with rb=2, ra=5, rt=7
instruction_odd = 32'b01011010101101010100101010100101;
rt_addr_even = 7'b0000101;
rt_addr_odd = 7'b0000101;
rt_addr_even_input = 128'hA7E5F23D6C9B1A4E5F2C7F1A5B3D6E9;
rt_addr_odd_input = 128'h1B3A5C7E9A2B4D6F8C9E2A4B6D8F1A5;
@(posedge clock) #1;
@(posedge clock) #8;
// Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
#100;
```

```
    $stop;
  end
endmodule
```

Register File Waveform:



Forwarding:

Forwarding File:

“Forward” module is a processor's functionality of managing the forwarding logic for the register values. This module is created to efficiently forward data within the processor to help prove that the correct data is propagated to subsequent pipeline stages when needed.

This module takes into account a few inputs including clock signals, reset signals, instructions from both even and odd pipeline stages, register values for different instructions, and flags indicating whether values will be written to registers. It will then evaluate these inputs to determine whether data is needed for forwarding and if so from which pipeline stage.

For each operand (such as 'ra', 'rb', 'rc', and 'rt_st') in both even and odd instructions, the module compared the source register of the current instruction with the destination register of any instruction in the even or odd pipeline stages. If there is a match and the corresponding instruction is written to the register, the module then forwards the data from the appropriate pipeline stage. This will help to ensure that the data is correctly forwarded based on the dependencies between the instructions in the pipeline. Additionally, the module handles cases where the forwarding is not necessary by defaulting to the input register values when no forwarding condition is met. Through these functions, the “Forward” module contributes to the

efficiency and performance of the processor by optimizing the data flow and minimizing the pipeline stalls.

Forwarding File Code:

```
/*************************************************************************************************************
 * Module: Forward
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *                          This module handles forwarding logic for register values in a processor.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - instruction_even: Instruction from even pipeline stage
 *   - instruction_odd: Instruction from odd pipeline stage
 *   - ra_even_input: Value of register A for even pipeline stage
 *   - rb_even_input: Value of register B for even pipeline stage
 *   - rc_even_input: Value of register C for even pipeline stage
 *   - ra_odd_input: Value of register A for odd pipeline stage
 *   - rb_odd_input: Value of register B for odd pipeline stage
 *   - rt_st_odd_input: Value of temporary register ST for odd pipeline stage
 *   - fw_even_wb_input: Pipe shift register of values ready to be forwarded for even stage
 *   - fw_odd_wb_input: Pipe shift register of values ready to be forwarded for odd stage
 *   - fw_addr_even_wb_input: Destinations of values to be forwarded for even stage
 *   - fw_addr_odd_wb_input: Destinations of values to be forwarded for odd stage
 *   - fw_write_even_wb_input: Flag indicating whether forwarded values will be written for even stage
 *   - fw_write_odd_wb_input: Flag indicating whether forwarded values will be written for odd stage
 *-------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - ra_even_fwd_output: Forwarded value of register A for even pipeline stage
 *   - rb_even_fwd_output: Forwarded value of register B for even pipeline stage
 *   - rc_even_fwd_output: Forwarded value of register C for even pipeline stage
 *   - ra_odd_fwd_output: Forwarded value of register A for odd pipeline stage
```

```
 *   - rb_odd_fwd_output: Forwarded value of register B for odd pipeline stage
 *   - rt_st_odd_fwd_output: Forwarded value of temporary registerrt_st for odd pipeline stage
 ************************************************************************************************/

module Forward (
    clock,
    reset,
    instruction_even,
    instruction_odd,
    ra_even_input,
    rb_even_input,
    rc_even_input,
    ra_odd_input,
    rb_odd_input,
    rt_st_odd_input,
    ra_even_fwd_output,
    rb_even_fwd_output,
    rc_even_fwd_output,
    ra_odd_fwd_output,
    rb_odd_fwd_output,
    rt_st_odd_fwd_output,
    fw_even_wb_input,
    fw_addr_even_wb_input,
    fw_write_even_wb_input,
    fw_odd_wb_input,
    fw_addr_odd_wb_input,
    fw_write_odd_wb_input
);

  input clock, reset;
  // Input instructions for both even and odd cycles, received from the decoder.
  input [0:31] instruction_even, instruction_odd;
  // Input values for 'ra', 'rb', 'rc' for both even and odd instructions, and 'rt_st' for odd instruction, retrieved from the Register Table.
  input [0:127] ra_even_input, rb_even_input, rc_even_input, ra_odd_input, rb_odd_input, rt_st_odd_input;
  // Pipeline data in shift registers, ready for forwarding
```

```
  input [6:0][0:127] fw_even_wb_input, fw_odd_wb_input;
  // Destination registers for data forwarding
  input [6:0][0:6] fw_addr_even_wb_input, fw_addr_odd_wb_input;
  // Flags indicating if forwarded values will be written to registers
  input [6:0] fw_write_even_wb_input, fw_write_odd_wb_input;
  // Updated input data after forwarding
  output logic [0:127] ra_even_fwd_output, rb_even_fwd_output, rc_even_fwd_output, ra_odd_fwd_output, rb_odd_fwd_output,
rt_st_odd_fwd_output;


  always_comb begin
    /* Evaluating 'ra' operand for the even instruction. If the source register of the even instruction
                        matches the destination register of any instruction in the even or odd pipeline stages and that
                        instruction is writing to the register, forward the data from the corresponding pipeline stage. */
    for (int i = 0; i < 7; ++i) begin
      if (instruction_even[18:24] == fw_addr_even_wb_input[i] && fw_write_even_wb_input[i] == 1)
        ra_even_fwd_output = fw_even_wb_input[i];
      else if (instruction_even[18:24] == fw_addr_odd_wb_input[i] && fw_write_odd_wb_input[i] == 1)
        ra_even_fwd_output = fw_odd_wb_input[i];
    end
    // If ra_even_fwd_output is unassigned, default to the value of ra_even_input.
    if (ra_even_fwd_output === 'bx) ra_even_fwd_output = ra_even_input;
    /* Evaluating 'rb' operand for the even instruction. If the source register of the even instruction
                        matches the destination register of any instruction in the even or odd pipeline stages and that
                        instruction is writing to the register, forward the data from the corresponding pipeline stage. */
    for (int i = 0; i < 7; ++i) begin
      if (instruction_even[11:17] == fw_addr_even_wb_input[i] && fw_write_even_wb_input[i] == 1)
        rb_even_fwd_output = fw_even_wb_input[i];
      else if (instruction_even[11:17] == fw_addr_odd_wb_input[i] && fw_write_odd_wb_input[i] == 1)
        rb_even_fwd_output = fw_odd_wb_input[i];
    end
    // If rb_even_fwd_output is unassigned, default to the value of rb_even_input.
    if (rb_even_fwd_output === 'bx) rb_even_fwd_output = rb_even_input;
    /* Evaluating 'rc' operand for the even instruction. If the source register of the even instruction
                        matches the destination register of any instruction in the even or odd pipeline stages and that
                        instruction is writing to the register, forward the data from the corresponding pipeline stage. */
```

```
for (int i = 0; i < 7; ++i) begin
  if (instruction_even[25:31] == fw_addr_even_wb_input[i] && fw_write_even_wb_input[i] == 1)
    rc_even_fwd_output = fw_even_wb_input[i];
  else if (instruction_even[25:31] == fw_addr_odd_wb_input[i] && fw_write_odd_wb_input[i] == 1)
    rc_even_fwd_output = fw_odd_wb_input[i];
end
// If rc_even_fwd_output is unassigned, assign it the value of rc_even_input.
if (rc_even_fwd_output === 'bx)  // Check if rc_even_fwd_output is not assigned
  rc_even_fwd_output = rc_even_input;
/* Evaluating 'rc' operand for the even instruction. If the source register of the even instruction
                matches the destination register of any instruction in the even or odd pipeline stages and that
                instruction is writing to the register, forward the data from the corresponding pipeline stage. */
for (int i = 0; i < 7; ++i) begin
  if (instruction_even[25:31] == fw_addr_even_wb_input[i] && fw_write_even_wb_input[i] == 1)
    rc_even_fwd_output = fw_even_wb_input[i];
  else if (instruction_even[25:31] == fw_addr_odd_wb_input[i] && fw_write_odd_wb_input[i] == 1)
    rc_even_fwd_output = fw_odd_wb_input[i];
end
// If rc_even_fwd_output is unassigned, default to the value of rc_even_input.
if (rc_even_fwd_output === 'bx) rc_even_fwd_output = rc_even_input;
/* Evaluating 'rb' operand for the odd instruction. If the source register of the odd instruction
                matches the destination register of any instruction in the even or odd pipeline stages and that
                instruction is writing to the register, forward the data from the corresponding pipeline stage. */
for (int i = 0; i < 7; ++i) begin
  if (instruction_odd[11:17] == fw_addr_even_wb_input[i] && fw_write_even_wb_input[i] == 1)
    rb_odd_fwd_output = fw_even_wb_input[i];
  else if (instruction_odd[11:17] == fw_addr_odd_wb_input[i] && fw_write_odd_wb_input[i] == 1)
    rb_odd_fwd_output = fw_odd_wb_input[i];
end
// If rb_odd_fwd_output is unassigned, default to the value of rb_odd_input.
if (rb_odd_fwd_output === 'bx) rb_odd_fwd_output = rb_odd_input;
/* Loop through each pipeline stage. If the destination register of the odd instruction matches the
                source register of any instruction in the even or odd pipeline stages and that instruction is
                writing to the register, forward the data from the corresponding pipeline stage. */
for (int i = 0; i < 7; ++i) begin
```

```
    if (instruction_odd[25:31] == fw_addr_even_wb_input[i] && fw_write_even_wb_input[i] == 1)
      rt_st_odd_fwd_output = fw_even_wb_input[i];
    else if (instruction_odd[25:31] == fw_addr_odd_wb_input[i] && fw_write_odd_wb_input[i] == 1)
      rt_st_odd_fwd_output = fw_odd_wb_input[i];
  end
  // If rt_st_odd_fwd_output is unassigned, assign it the value of rt_st_odd_input.
  if (rt_st_odd_fwd_output === 'bx) rt_st_odd_fwd_output = rt_st_odd_input;
 end
endmodule
```

Forwarding Test Bench:

“Forward” testbench module is important for validating the functionality of the Forward module. Forwarding logic enables the direct transfer of data between pipeline stages, which helps mitigate stalls caused by data errors and also enhances the performance of the processor.

This testbench has various inputs that are needed to simulate the behavior of the processor during different stages of the pipeline. These inputs include clock signals, reset signals, instructions from both even and odd pipeline stages, register values, destination register addresses, values to be written to destination registers, and flags indicating whether instructions will be written to registers. By configuring these inputs and learning for the outputs, this testbench facilitates thorough testing of the forward modules functionality under a board range of scenarios. This will help ensure that the forwarded data operates as expected across different pipeline stages.

Through the execution of the testbench, scenarios will be simulated, read and write operations will be performed to the same register address by even and odd pipeline stages. This will allow for validation of forwarding logic effectiveness in resolving potential data errors. By using these scenarios and analyzing the results, this testbench enables verification of the forwarding logic implementation in the processor to behave as intended, helping with the reliability of the processor's design.

Forwarding Test Bench Code:

```
/**********************************************************************************************************
 * Module: Forward Testbench
 * Author: Noah Merone
 *---------------------------------------------------------------------------------------------------------
```

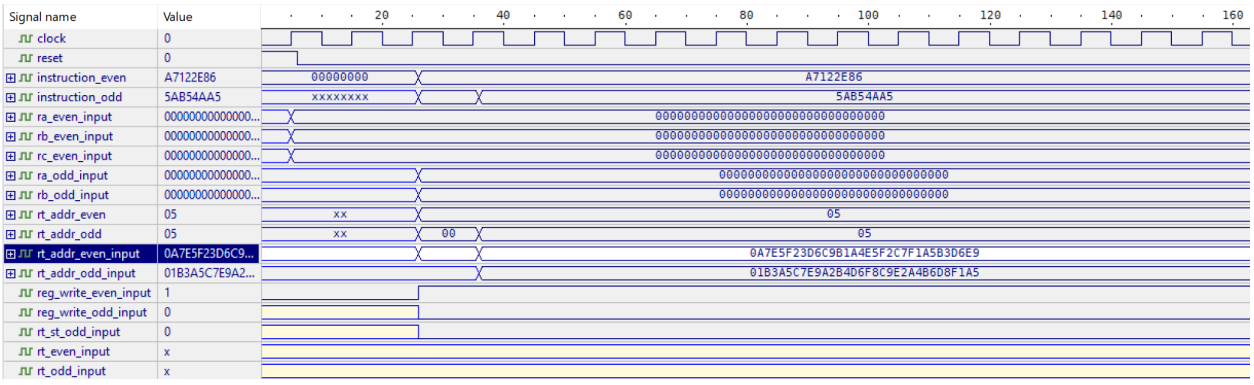
```
* Description:
*     This testbench module verifies the functionality of the Register File module, which handles register
*     file operations including forwarding logic in a processor. Forwarding logic allows data to be passed
*     directly between pipeline stages, reducing stalls caused by data hazards and enhancing processor performance.
*-----------------------------------------------------------------------------------------------------
* Inputs:
*   - clock: Clock signal
*   - reset: Reset signal
*   - instruction_even: Instruction from even pipeline stage
*   - instruction_odd: Instruction from odd pipeline stage
*   - ra_even_input: Value of register A for even pipeline stage
*   - rb_even_input: Value of register B for even pipeline stage
*   - rc_even_input: Value of register C for even pipeline stage
*   - ra_odd_input: Value of register A for odd pipeline stage
*   - rb_odd_input: Value of register B for odd pipeline stage
*   - rt_st_odd_input: Value of temporary register rt_st for odd pipeline stage
*   - rt_addr_even_input: Destination register address to write to for even pipeline stage
*   - rt_addr_odd_input: Destination register address to write to for odd pipeline stage
*   - rt_even_input: Value to write to the destination register for the even pipeline stage
*   - rt_odd_input: Value to write to the destination register for the odd pipeline stage
*   - reg_write_even_input: Flag indicating whether the even instruction will write to a register
*   - reg_write_odd_input: Flag indicating whether the odd instruction will write to a register
*-----------------------------------------------------------------------------------------------------
* Outputs:
*   - ra_even_input: Value of register A for the even pipeline stage
*   - rb_even_input: Value of register B for the even pipeline stage
*   - rc_even_input: Value of register C for the even pipeline stage
*   - ra_odd_input: Value of register A for the odd pipeline stage
*   - rb_odd_input: Value of register B for the odd pipeline stage
*   - rt_st_odd_input: Value of temporary register rt_st for the odd pipeline stage
*-----------------------------------------------------------------------------------------------------
* Internal Signals:
*   - registers: Register file storing register values
*   - i: 8-bit counter for the reset loop
*****************************************************************************************************/
```

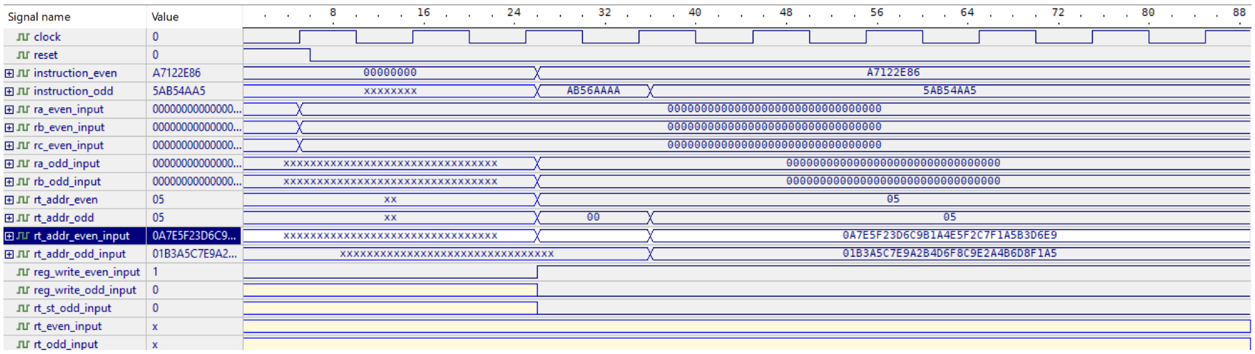
```
module Forward_TB ();
  logic clock, reset;

  // Input instructions for both even and odd cycles, received from the decoder
  logic [0:31] instruction_even, instruction_odd;
  // Input values for 'ra', 'rb', 'rc' for both even and odd instructions, retrieved from the Register Table
  logic [0:127] ra_even_input, rb_even_input, rc_even_input, ra_odd_input, rb_odd_input;
  // Destination registers for even and odd instructions
  logic [0:6] rt_addr_even, rt_addr_odd;
  // Values to be written to destination registers for even and odd instructions
  logic [0:127] rt_addr_even_input, rt_addr_odd_input;
  // Flags indicating if the even and odd instructions will write to the register table
  logic reg_write_even_input, reg_write_odd_input;
  // Indicates whether the data from the Store will be written to the odd pipe register file input
  logic rt_st_odd_input;
  // Indicates whether the data from the even pipe register file will be written to the odd pipe register file input
  logic rt_even_input;
  // Indicates whether the data from the odd pipe register file will be written to the odd pipe register file input
  logic rt_odd_input;

  Register_File dut (
      clock,
      reset,
      instruction_even,
      instruction_odd,
      ra_even_input,
      rb_even_input,
      rc_even_input,
      ra_odd_input,
      rb_odd_input,
      rt_st_odd_input,
      rt_addr_even_input,
      rt_addr_odd_input,
      rt_even_input,
```

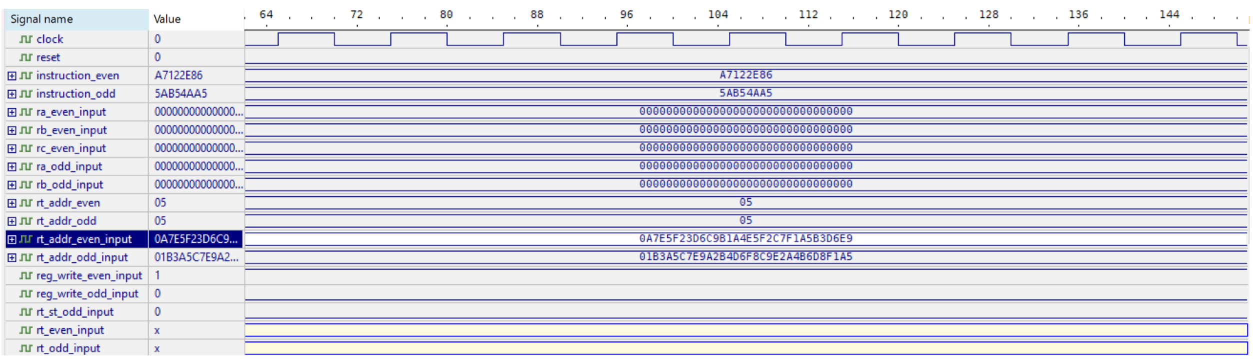
```
    rt_odd_input,
    reg_write_even_input,
    reg_write_odd_input
);

initial clock = 0;

always begin
  #5 clock = ~clock;
end

initial begin
  reset = 1;
  reg_write_even_input = 0;
  instruction_even = 32'h00000000;
  #6;
  // Enable unit at 11ns
  reset = 0;
  @(posedge clock);
  #1;
  @(posedge clock);
  #1;
  reg_write_even_input = 1;
  // shlh instruction with rb=3, ra=4, rt=5
  instruction_even = 32'b10100111000100100010111010000110;
  rt_addr_even = 7'b000101;
  rt_addr_even_input = 128'h3F8A1B9E2C7F1A5B3D6E9C2F5A8B1C4;
  // lnop instruction
  instruction_odd = 32'b10101011010101101010101010101010;
  reg_write_odd_input = 0;
  rt_addr_odd = 7'b0000000;
  // Writing the value rt_addr_even_input to address 7
  @(posedge clock) #1;
  // Attempting to read the value at address 7 for odd pipe while even pipe is also writing back to the same
  // register file (7). Odd pipe is able to read the value from reg 7 before even pipe writes back a new value to it.
```

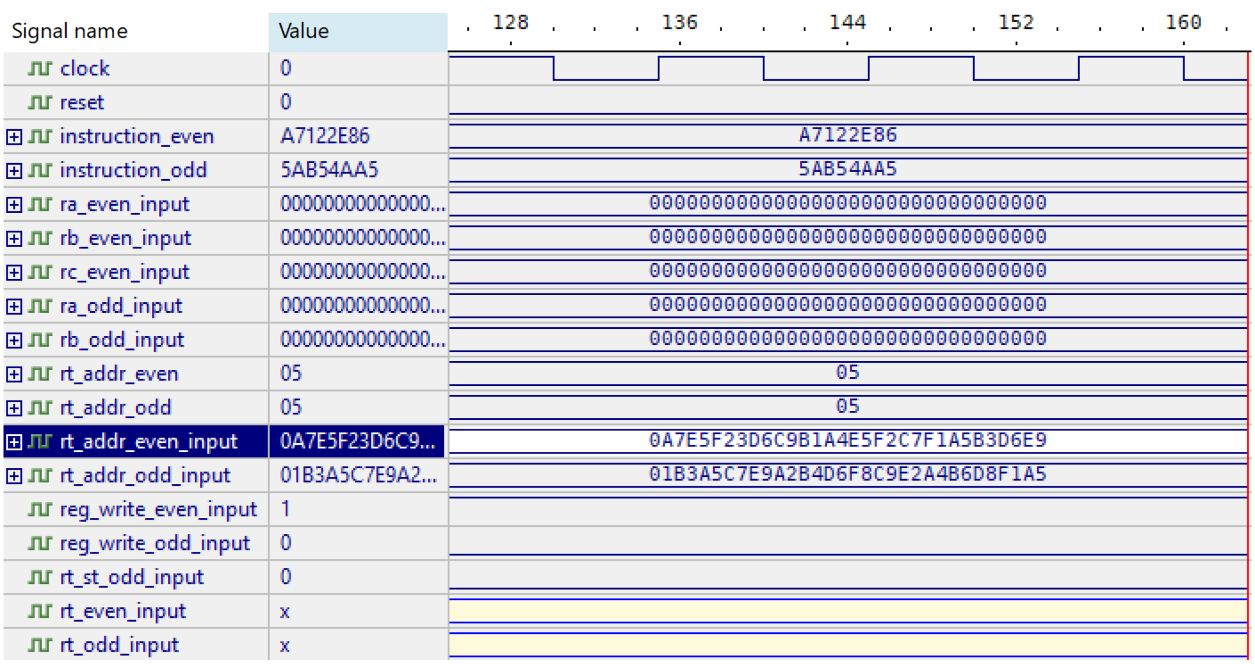
```
    reg_write_even_input = 1;
    // shlh instruction with rb=3, ra=4, rt=5
    instruction_even = 32'b10100111000100100001011101000011O;
    // shlqi instruction with rb=2, ra=5, rt=7
    instruction_odd = 32'b01011010101101010100101010100101;
    rt_addr_even = 7'b0000101;
    rt_addr_odd = 7'b0000101;
    rt_addr_even_input = 128'hA7E5F23D6C9B1A4E5F2C7F1A5B3D6E9;
    rt_addr_odd_input = 128'h1B3A5C7E9A2B4D6F8C9E2A4B6D8F1A5;
    @(posedge clock) #1;
    @(posedge clock) #8;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #100;
    $stop;
  end
endmodule
```

Forwarding Waveform:








Signal name
Value
128
136
144
152
160
clock 0
reset 0
instruction_even A7122E86
instruction_odd 5AB54AA5
ra_even_input 0000000000000...
rb_even_input 0000000000000...
rc_even_input 0000000000000...
ra_odd_input 0000000000000...
rb_odd_input 0000000000000...
rt_addr_even 05
rt_addr_odd 05
rt_addr_even_input 0A7E5F23D6C9...
rt_addr_odd_input 01B3A5C7E9A2...
reg_write_even_input 1
reg_write_odd_input 0
rt_st_odd_input 0
rt_even_input x
rt_odd_input x
A7122E86
5AB54AA5
00000000000000000000000000000000
05
0A7E5F23D6C9B1A4E5F2C7F1A5B3D6E9
01B3A5C7E9A2B4D6F8C9E2A4B6D8F1A5

Branch:

Branch:

The “Branch” module is an important part for determining the next program counter and handling the forwarding of register values. Branch instructions are crucial for altering the flow of program execution based on certain calculations or even conditions. This module ensures the correctness of instructions, optimizing the processor’s performance and efficiency.

This module receives various inputs including clock signals, reset signals, decoded opcodes, instruction formats, register values, immediate values, and flags indicating whether instructions will write to the register table. Based on these inputs, the module calculates the new program counter for branch instructions and even manages the forwarding of the register values. It also will determine if the branch is taken or not based on certain conditions like zero/non zero, or specific opcode patterns, and will update the program counter accordingly. Additionally, the module handles scenarios where branch instructions need to be disabled if they are initial instructions in a pair to ensure proper sequencing of instructions and proper execution of instructions.

In the design of the “Branch” module, it is important to create an implementation that seamlessly executes instructions within the processor. This will contribute to the functionality and performance of the processor architecture. By successfully determining the next program counter and even managing the register forwarding, this module enables efficient branching and control flow within the processor. This will enhance execution with a wide range of instructions and programs.

Branch Code:

```
/*************************************************************************************************************
 * Module: Branch
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *     This module handles branch instructions in a processor, determining the next program counter and
 *     managing the forwarding of register values.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Decoded opcode truncated based on instr_format
 *   - instr_format: Format of the instruction, used with op_code and imm_value
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Value of source register A
 *   - src_reg_b: Value of source register B
 *   - store_reg: Value of the store register
 *   - imm_value: Immediate value truncated based on instr_format
 *   - enable_reg_write: Flag indicating whether current instruction writes to RegTable
 *   - program_counter_input: Program counter from IF stage
 *   - initial_: 1 if the initial_ instruction in pair; used for determining the order of a branch
 *-------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Output value of Stage 3
```

```
 *   - wb_reg_addr: Destination register for wb_data
 *   - wb_enable_reg_write: Will wb_data write to RegTable
 *   - program_counter_wb: New program counter for branch
 *   - branch_is_taken: Was the branch taken?
 *   - disable_branch: If the branch is taken and branch instruction is initial_ in pair, kill twin instruction
 *-------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - rt_delay: Staging register for calculated values
 *   - rt_addr_delay: Destination register for wb_data
 *   - reg_write_delay: Will wb_data write to RegTable
 *   - pc_delay: Staging register for PC
 *   - branch_delay: Was the branch taken?
 *******************************************************************************************************/

module Branch (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    store_reg,
    imm_value,
    enable_reg_write,
    program_counter_input,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    program_counter_wb,
    branch_is_taken,
    initial_,
    disable_branch
);

  input clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  input [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  input [2:0] instr_format;
  // Destination register address
  input [0:6] dest_reg_addr;
  // Values of source registers
  input [0:127] src_reg_a, src_reg_b, store_reg;
  // Immediate value, truncated based on instruction format
  input [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  input enable_reg_write;
  // Program counter from IF stage
  input [7:0] program_counter_input;
  // 1 if initial_ instruction in pair; used for determining order of branch
  input initial_;

  // Write Back Stage
  // Output value of Stage 3
  output logic [0:127] wb_data;
  // Destination register for write back data
  output logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
```

```
output logic wb_enable_reg_write;
// New program counter for branch
output logic [7:0] program_counter_wb;
// Was branch taken?
output logic branch_is_taken;
// If branch is taken and branch instruction is initial_ in pair, kill twin instruction
output logic disable_branch;

//Internal Signals
// Staging register for calculated values
logic [0:127] rt_delay;
// Destination register for write back data
logic [0:6] rt_addr_delay;
// Flag indicating if the write back data will be written to the Register Table
logic reg_write_delay;
// Staging register for Program Counter
logic [7:0] pc_delay;
// Flag indicating if the branch was taken
logic branch_delay;

always_ff @(posedge clock) begin
  if (reset == 1) begin
    wb_data <= 0;
    wb_reg_addr <= 0;
    wb_enable_reg_write <= 0;
    program_counter_wb <= 0;
    branch_is_taken <= 0;
  end else begin
    wb_data <= rt_delay;
    wb_reg_addr <= rt_addr_delay;
    wb_enable_reg_write <= reg_write_delay;
    program_counter_wb <= pc_delay;
    branch_is_taken <= branch_delay;
  end
end

always_comb begin
  // Case when the instruction format is 0 and opcode is 0: No Operation (Load)
  if (instr_format == 0 && op_code == 0) begin
    rt_delay = 0;
    rt_addr_delay = 0;
    reg_write_delay = 0;
    pc_delay = 0;
    branch_delay = 0;
  end else begin
    rt_addr_delay   = dest_reg_addr;
    reg_write_delay = enable_reg_write;
    if (branch_is_taken) begin
      rt_delay = 0;
      rt_addr_delay = 0;
      reg_write_delay = 0;
      pc_delay = 0;
      branch_delay = 0;
    end else if (instr_format == 0) begin
      case (op_code)
        //bi: Branch Indirect
        11'b00110101000: begin
          pc_delay = src_reg_a;
          reg_write_delay = 1'b0;
          branch_delay = 1'b1;
        end
```

```
                                // biz: Branch Indirect If Zero
                                11'b00100101000: begin
                                        integer target_address;
                                  branch_delay = 1;
                                  if (store_reg[0:3] != 0) begin
                                    target_address = store_reg & (program_counter_input + 4);
                                  end else if ((src_reg_a[4] == 1 && src_reg_a[5] == 0) || (src_reg_a[4] == 0 && src_reg_a[5] == 1)) begin
                                    target_address = (src_reg_a[0:3] << 2) & 32'hFFFFFFFC;
                                  end
                                  pc_delay = target_address;
                                end
                                // bihz: Branch Indirect If Zero Halfword
                                11'b00100101010: begin
                                        integer target_address;
                                  branch_delay = 1;
                                  if (store_reg[2:3] != 2'b00) begin
                                    target_address = store_reg & (program_counter_input + 4);
                                  end else if ((src_reg_a[4] == 1'b1 && src_reg_a[5] == 1'b0) || (src_reg_a[4] == 1'b0 && src_reg_a[5] ==
1'b1)) begin
                                    target_address = (src_reg_a[0:3] << 2) & 32'hFFFFFFFC;
                                  end
                                  pc_delay = target_address;
                                end

      default begin
        rt_delay = 0;
        rt_addr_delay = 0;
        reg_write_delay = 0;
        pc_delay = 0;
        branch_delay = 0;
      end
    endcase
  end else if (instr_format == 5) begin
    case (op_code[2:10])
                                // brsl: Branch Relative and Set Link
                                9'b001100110: begin
                                        integer target_address;
                                  rt_delay[0:31] = program_counter_input + 1;
                                  rt_delay[32:127] = 0;
                                  target_address = $signed(imm_value[2:17]);
                                  pc_delay = program_counter_input + target_address;
                                  branch_delay = 1;
                                end
                                // br: Branch Relative
                                9'b001100100: begin
                                        integer target_address;
                                  target_address = $signed(imm_value[2:17]);
                                  pc_delay = program_counter_input + target_address;
                                  branch_delay = 1;
                                end
                                // bra: Branch Absolute
                                9'b001100000: begin
                                        integer target_address;
                                  target_address = imm_value[2:17];
                                  pc_delay = target_address;
                                  branch_delay = 1;
                                end
                                // brasl: Branch Absolute and Set Link
                                9'b001100010: begin
                                        integer new_pc_value;
```

```
                    // Calculate the new Program Counter (PC) value by adding 4 to the current PC and masking the lower 2 bits.
Store the result in the lower 32 bits of rt_delay.
                    rt_delay[0:31] = (program_counter_input + 4) & 32'hFFFF_FFFC;
                    // Set the upper 96 bits of rt_delay to 0
                    rt_delay[32:127] = 0;
                    // Calculate the new Program Counter (PC) value by masking the lower 2 bits of the immediate value. Store
the result in pc_delay.
                    new_pc_value = {16'b0, imm_value[2:17]} & 32'hFFFF_FFFC;
                    pc_delay = new_pc_value;
                    branch_delay = 1;
                end
                // Handling the "Branch If Not Zero Halfword" operation (brnz)
                9'b001000110: begin
                        integer offset;
                    offset = (initial_) ? 2 : 1;
                    pc_delay = program_counter_input - offset + $signed(imm_value[2:17]);
                    branch_delay = (store_reg[0:15] != 16'h0000) ? 1'b1 : 1'b0;
                end
                // binz: Branch Indirect If Not Zero
                11'b00100101001: begin
                        integer target_address;
                    branch_delay = 1;
                    if (store_reg[0:3] != 0) begin
                        target_address = (src_reg_a[0:3] << 2) & 32'hFFFFFFFC;
                    end else begin
                        target_address = store_reg & (program_counter_input + 4);
                    end
                    pc_delay = target_address;
                end
                // binz: Branch Indirect If Not Zero Halfword
                11'b00100101011: begin
                        integer target_address;
                    branch_delay = 1;
                    if (store_reg[2:3] != 2'b00) begin
                        target_address = store_reg & (program_counter_input + 4);
                        pc_delay = target_address;
                    end else if ((src_reg_a[4] == 1'b1 && src_reg_a[5] == 1'b0) || (src_reg_a[4] == 1'b0 && src_reg_a[5] ==
1'b1)) begin
                        target_address = (src_reg_a[0:3] << 2) & 32'hFFFFFFFC;
                        pc_delay = target_address;
                    end
                end
                // brhz: Branch If Zero Halfword
                9'b001000100: begin
                        integer offset;
                    offset = (store_reg[32:33] == 2'b00) ? $signed(imm_value[2:17]) : 4;
                    branch_delay = 1;
                    pc_delay = (program_counter_input + offset) & 32'hFFFFFFFC;
                end
                // brnz: Branch If Not Zero Word
                9'b001000010: begin
                        integer offset;
                    offset = (initial_) ? 2 : 1;
                    pc_delay = program_counter_input - offset + $signed(imm_value[2:17]);
                    branch_delay = (store_reg != 0) ? 1'b1 : 1'b0;
                end
                // brz: Branch If Zero Word
                9'b001000000: begin
                        integer offset;
                    offset = (initial_) ? 2 : 1;
                    pc_delay = program_counter_input - offset + $signed(imm_value[2:17]);
```

```
                                    branch_delay = (store_reg == 0) ? 1'b1 : 1'b0;
                                end
        default begin
          rt_delay = 0;
          rt_addr_delay = 0;
          reg_write_delay = 0;
          pc_delay = 0;
          branch_delay = 0;
        end
      endcase
    end else begin
      rt_delay = 0;
      rt_addr_delay = 0;
      reg_write_delay = 0;
      pc_delay = 0;
      branch_delay = 0;
    end
  end
  if (branch_delay == 1 && initial_ == 1) disable_branch = 1;
  else disable_branch = 0;
 end
endmodule
```

Branch Test Bench:

The “Branch” test bench module is meant to verify the functionality of the “Branch” module by providing simulation while also monitoring the outputs. This testbench is important to ensure that the branch instructions within the processor function correctly by stimulating different scenarios and learning from the responses of the module.

This testbench module receives inputs like clock signals, reset signals, decoded opcodes, instruction formats, register values, immediate values, and flags indicating whether instructions will write to the register table. It will then simulate the “Branch” module by providing different combinations of inputs, including different operation code patterns and instruction formats. This will help to evaluate the module and how it computes the new program counter for branch instructions and manages register forwarding. Also, this testbench module will monitor outputs such as the new program counter, whether a branch is or isn't taken, and whether a branch is to be terminated, allowing for better verification of the “Branch” modules functionality.

By testing the “Branch” module under unique conditions and scenarios, we can ensure that the module is operating correctly and validate its effectiveness in managing branch instructions in the processor. This verification will help with the overall reliability of the architecture for real world applications.

Branch Test Bench Code:

```
/*****************************************************************************************************************
 * Module: Branch Testbench
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------------------
 * Description:
 *     This testbench module verifies the functionality of the Branch module by providing stimulus and
 *     monitoring the outputs.
 *-----------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Decoded opcode truncated based on instr_format
 *   - instr_format: instr_format of the instruction, used with op_code and imm_value
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Value of source register A
 *   - src_reg_b: Value of source register B
 *   - store_reg: Value of the store register
 *   - imm_value: Immediate value truncated based on instr_format
 *   - enable_reg_write: Flag indicating whether current instruction writes to RegTable
 *   - program_counter_input: Program counter from IF stage
 *   - initial_: 1 if the initial_ instruction in pair; used for determining the order of a branch
 *-----------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Output value of Stage 3
 *   - wb_reg_addr: Destination register for wb_data
 *   - wb_enable_reg_write: Will wb_data write to RegTable
 *   - program_counter_wb: New program counter for branch
 *   - branch_is_taken: Was the branch taken?
 *   - disable_branch: If the branch is taken and branch instruction is initial_ in pair, kill twin instruction
 *-----------------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - rt_delay: Staging register for calculated values
 *   - rt_addr_delay: Destination register for wb_data
 *   - reg_write_delay: Will wb_data write to RegTable
```

```
*   - pc_delay: Staging register for PC
*   - branch_delay: Was the branch taken?
*****************************************************************************************************/

module Branch_TB ();
  logic clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction instr_format
  logic [0:10] op_code;
  // instr_format of instruction, used with opcode and immediate value
  logic [ 2:0] instr_format;
  // Destination register address
  logic [ 0:6] dest_reg_addr;
  // Values of source registers
  logic [0:127] src_reg_a, src_reg_b, store_reg;
  // Immediate value, truncated based on instruction instr_format
  logic [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  logic enable_reg_write;
  // Program counter from IF stage
  logic [7:0] program_counter_input;

  // Write Back Stage
  // Output value of Stage 3
  logic [0:127] wb_data;
  // Destination register for write back data
  logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  logic wb_enable_reg_write;
  // New program counter for branch
  logic [7:0] program_counter_wb;
  // Indicates whether a branch is taken
  logic branch_is_taken;
  // Indicates the initial state or condition
```

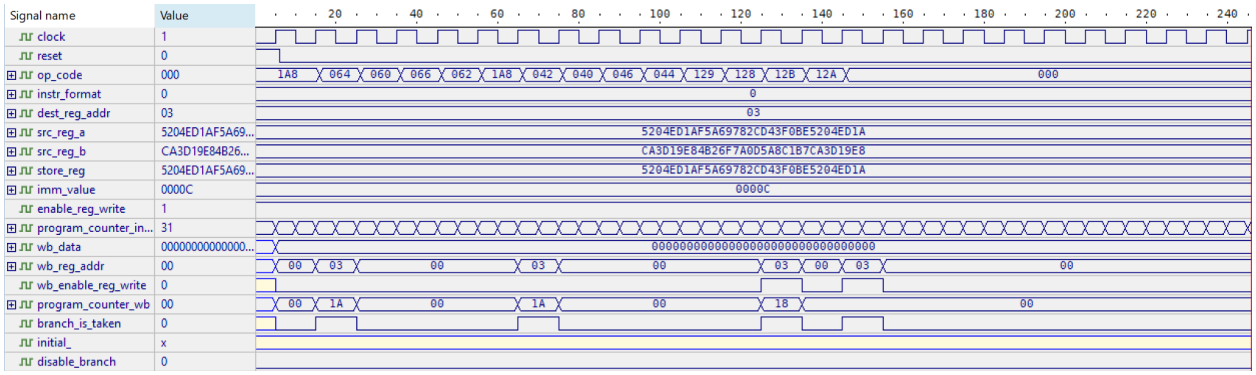
```
logic initial_;
// Indicates whether a branch is to be terminated
logic disable_branch;

Branch dut (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    store_reg,
    imm_value,
    enable_reg_write,
    program_counter_input,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    program_counter_wb,
    branch_is_taken,
    initial_,
    disable_branch
);

// Set the initial state of the clock to zero
initial clock = 0;

// Toggle the clock value every 5 time units to simulate oscillation
always begin
  #5 clock = ~clock;
  program_counter_input = program_counter_input + 1;
end

initial begin
```

```
reset = 1;
instr_format = 3'b000;
// Set the opcode for the Branch Indirect instruction (bi)
op_code = 11'b00110101000;
// Set the destination register address to $r3
dest_reg_addr = 7'b0000011;
// Set the value of source register A
src_reg_a = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
// Set the value of source register B
src_reg_b = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
// Set the value of the store register
store_reg = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
imm_value = 12;
enable_reg_write = 1;
program_counter_input = 0;
#6;
// At 11ns, disable the reset, enabling the unit
reset = 0;
@(posedge clock);
#1;
// Branch Relative (br)
op_code = 9'b001100100;
@(posedge clock);
#1;
// Branch absolute (bra) operation
op_code = 9'b001100000;
@(posedge clock);
#1;
// Branch Relative and Set Link (brsl)
op_code = 9'b001100110;
@(posedge clock);
#1;
// Branch Absolute and Set Link (brasl)
op_code = 9'b001100010;
@(posedge clock);
```

```
#1;
// Branch Indirect (bi)
op_code = 11'b00110101000;
@(posedge clock);
#1;
// Branch If Not Zero Word (brnz)
op_code = 9'b001000010;
@(posedge clock);
#1;
// Branch If Zero Word (brz)
op_code = 9'b001000000;
@(posedge clock);
#1;
// Branch If Not Zero Halfword (brnzh)
op_code = 9'b001000110;
@(posedge clock);
#1;
// Branch If Zero Halfword (brzh)
op_code = 9'b001000100;
@(posedge clock);
#1;
// Branch Indirect If Not Zero (binz)
op_code = 11'b00100101001;
@(posedge clock);
#1;
// Branch Indirect If Zero (biz)
op_code = 11'b00100101000;
@(posedge clock);
#1;
// Branch Indirect If Not Zero Halfword (binzh)
op_code = 11'b00100101011;
@(posedge clock);
#1;
// Branch Indirect If Zero Halfword (bihz)
op_code = 11'b00100101010;
```

```
    @(posedge clock);
    #1;
    // Set the opcode for the No Operation (nop) instruction
    op_code = 0;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #100;
    op_code = 11'b00000000000;
    $stop;
  end
endmodule
```

Branch Waveform:



Byte:

Byte:

The “Byte” module facilitates the execution of byte level instructions in a processor, helping with the overall functionality of the system. This module operates by receiving inputs including clock signals, reset signals, decoded opcodes, instruction formats, register values, immediate values, and flags indicating whether instructions will be written to the register table. Based on these inputs, the “Byte” module simulates the execution of byte level instructions and provides calculated results to the write back stage for further processing.

The functionality of the “Byte” module encompasses the following: counting ones in bytes, computing the average of bytes, finding absolute differences between bytes, and summing bytes into halfwords. These operations are implemented using combinational and sequential logic in the module, allowing for processing of byte level data. Additionally, the module incorporates delay elements to ensure synchronization and correctness of handled data propagation across clock cycles. This helps to enhance the reliability of the byte level instruction execution process.

Through calculation of byte level data, the “Byte" module enables the processor to perform byte level operations, which expand its capabilities in handling a wide range of computational tasks. This module's integration into the processor's architecture enhances its versatility and will contribute to the performance and efficiency of the processor.

Byte Code:

```
/*************************************************************************************************************
 * Module: Byte
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *     This module simulates the execution of byte-level instructions, providing calculated results
 *     to the Write Back stage.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Decoded opcode truncated based on format
 *   - instr_format: Format of the instruction, used with op_code and imm_value
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Value of source register A
 *   - src_reg_b: Value of source register B
 *   - imm_value: Immediate value truncated based on format
 *   - enable_reg_write: Flag indicating whether current instruction writes to RegTable
 *   - branch_is_taken: Was branch taken?
 *-------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Output value of Stage 3
 *   - wb_reg_addr: Destination register for wb_data
 *   - wb_enable_reg_write: Will wb_data write to RegTable
 *   - delayed_rt_addr: Destination register for wb_data, delayed by one clock cycle
 *   - delayed_enable_reg_write: Will wb_data write to RegTable, delayed by one clock cycle
 *-------------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - delayed_rt_data: Staging register for calculated values, delayed by one clock cycle
 *************************************************************************************************************/

module Byte (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
```

```
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

  input clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  input [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  input [2:0] instr_format;
  // Destination register address
  input [0:6] dest_reg_addr;
  // Values of source registers
  input [0:127] src_reg_a, src_reg_b;
  // Immediate value, truncated based on instruction format
  input [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  input enable_reg_write;
  // Was branch taken?
  input branch_is_taken;

  // Write Back Stage
  // Output value of Stage 3
  output logic [0:127] wb_data;
  // Destination register for write back data
  output logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic wb_enable_reg_write;

  // Internal Signals
  // Staging register for calculated values
  logic [3:0][0:127] delayed_rt_data;
  // Destination register for write back data
  output logic [3:0][0:6] delayed_rt_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic [3:0] delayed_enable_reg_write;

  // Temporary variable used as an incrementing counter
  logic [3:0] temporary_variable;

  always_comb begin
    wb_data = delayed_rt_data[2];
    wb_reg_addr = delayed_rt_addr[2];
    wb_enable_reg_write = delayed_enable_reg_write[2];
  end

  always_ff @(posedge clock) begin
    if (reset == 1) begin
      delayed_rt_data[3] <= 0;
      delayed_rt_addr[3] <= 0;
      delayed_enable_reg_write[3] <= 0;
      for (int i = 0; i < 3; i = i + 1) begin
        delayed_rt_data[i] <= 0;
        delayed_rt_addr[i] <= 0;
```

```
    delayed_enable_reg_write[i] <= 0;
  end
end else begin
  delayed_rt_data[3] <= delayed_rt_data[2];
  delayed_rt_addr[3] <= delayed_rt_addr[2];
  delayed_enable_reg_write[3] <= delayed_enable_reg_write[2];
  delayed_rt_data[2] <= delayed_rt_data[1];
  delayed_rt_addr[2] <= delayed_rt_addr[1];
  delayed_enable_reg_write[2] <= delayed_enable_reg_write[1];
  delayed_rt_data[1] <= delayed_rt_data[0];
  delayed_rt_addr[1] <= delayed_rt_addr[0];
  delayed_enable_reg_write[1] <= delayed_enable_reg_write[0];
  //nop : No Operation (Load)
  if (instr_format == 0 && op_code == 0) begin
    delayed_rt_data[0] <= 0;
    delayed_rt_addr[0] <= 0;
    delayed_enable_reg_write[0] <= 0;
  end else begin
    delayed_rt_addr[0] <= dest_reg_addr;
    delayed_enable_reg_write[0] <= enable_reg_write;
    if (branch_is_taken) begin
      delayed_rt_data[0] <= 0;
      delayed_rt_addr[0] <= 0;
      delayed_enable_reg_write[0] <= 0;
    end else if (instr_format == 0) begin
      case (op_code)
        //cntb : Count Ones in Bytes
        11'b01010110100: begin
          automatic int i;
          for (i = 0; i < 16; i = i + 1) begin
            automatic int temporary_variable = 0;
            automatic int j;
            for (j = 0; j < 8; j = j + 1) begin
              if (src_reg_a[(i*8)+j] == 1'b1) temporary_variable = temporary_variable + 1;
            end
            delayed_rt_data[0][(i*8)+:8] <= temporary_variable;
          end
        end
        //avgb : Average Bytes
        11'b00011010011: begin
          automatic int i = 0;
          while (i < 16) begin
            automatic logic [9:0] temp_a = {2'b00, src_reg_a[(i*8)+:8]};
            automatic logic [9:0] temp_b = {2'b00, src_reg_b[(i*8)+:8]};
            delayed_rt_data[0][(i*8)+:8] <= (temp_a + temp_b + 1) >> 1;
            i = i + 1;
          end
        end
        //absdb : Absolute Differences of Bytes
        11'b00001010011: begin
          automatic int i = 0;
          while (i < 16) begin
            automatic logic [7:0] temp_a = src_reg_a[(i*8)+:8];
            automatic logic [7:0] temp_b = src_reg_b[(i*8)+:8];
            delayed_rt_data[0][(i*8) +: 8] <= (temp_a > temp_b) ? (temp_a - temp_b) : (temp_b - temp_a);
            i = i + 1;
          end
        end
        //sumb : Sum Bytes into Halfwords
        11'b01001010011: begin
          automatic int i = 0;
```

```
            while (i < 4) begin
              automatic int sum_b = 0;
              automatic int sum_a = 0;
              automatic int j = 0;
              while (j < 4) begin
                sum_b += $signed(src_reg_b[(i*32)+(j*8)+:8]);
                sum_a += $signed(src_reg_a[(i*32)+(j*8)+:8]);
                j = j + 1;
              end
              delayed_rt_data[0][(i*32)+:16] <= sum_b;
              delayed_rt_data[0][(i*32)+16+:16] <= sum_a;
              i = i + 1;
            end
          end
          default begin
            delayed_rt_data[0] <= 0;
            delayed_rt_addr[0] <= 0;
            delayed_enable_reg_write[0] <= 0;
          end
        endcase
      end else begin
        delayed_rt_data[0] <= 0;
        delayed_rt_addr[0] <= 0;
        delayed_enable_reg_write[0] <= 0;
      end
    end
  end
endmodule
```

Byte Test Bench:

The “Byte” testbench module serves as a test for the correctness of the Byte module within the processor design. Using simulation on the Byte module, we can monitor the outputs to ensure they align with what's expected for the behavior. This testbench module simulates the execution of byte level instructions by supplying various inputs like clock signals, reset signals, decoded opcodes, instruction formats, register values, immediate values, and flags indicating whether instructions will write to the register table.

During simulation, the “Byte” test bench module helps in the execution of different byte level operations, including: counting ones in bytes, computing averages of bytes, finding absolute differences between bytes, and summing bytes into halfwords. It achieves this by setting the opcodes for each operation and providing appropriate register values and control signals. The module then will observe the outputs from the Byte module, including: write back data, destination register addresses, and flags indicating write enable signals. By comparing these outputs with the expected results, this testbench can verify the correctness of the Byte module.

With testing and verification by the “Byte” testbench module, we can gain insight into the functionality of the Byte module in the processor design. This testing helps to ensure correctness and accuracy to perform byte level operations. This will help with the performance of the overall system.

Byte Test Bench Code:

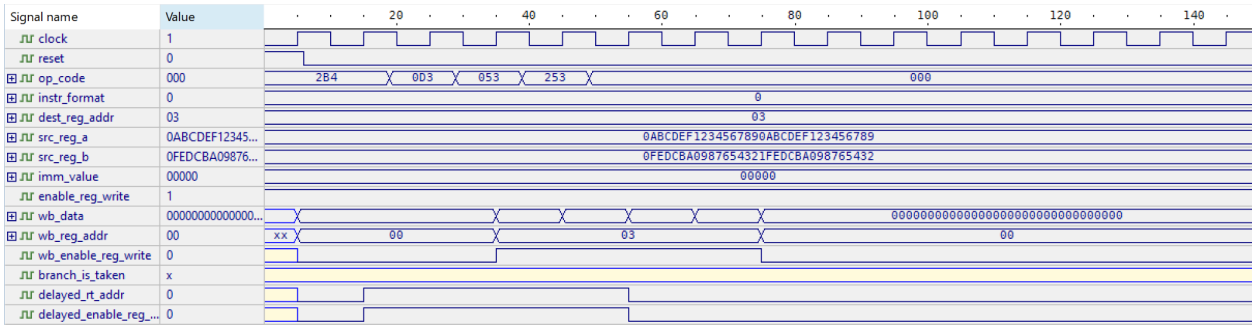
```
/*****************************************************************************************************************
 * Module: Byte Testbench
 * Author: Noah Merone
 *---------------------------------------------------------------------------------------------------------------
 * Description:
 *    This testbench module verifies the functionality of the Byte module by providing stimulus and
 *    monitoring the outputs.
 *---------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Decoded opcode truncated based on format
 *   - instr_format: Format of the instruction, used with op_code and imm_value
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Value of source register A
 *   - src_reg_b: Value of source register B
 *   - imm_value: Immediate value truncated based on format
 *   - enable_reg_write: Flag indicating whether current instruction writes to RegTable
 *   - branch_is_taken: Was branch taken?
 *---------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Output value of Stage 3
 *   - wb_reg_addr: Destination register for wb_data
 *   - wb_enable_reg_write: Will wb_data write to RegTable
 *   - delayed_rt_addr: Destination register for wb_data, delayed by one clock cycle
 *   - delayed_enable_reg_write: Will wb_data write to RegTable, delayed by one clock cycle
 *---------------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - delayed_rt_data: Staging register for calculated values, delayed by one clock cycle
 *****************************************************************************************************************/
```

```
module Byte_TB ();
  logic clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  logic [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  logic [ 2:0] instr_format;
  // Destination register address
  logic [ 0:6] dest_reg_addr;
  // Values of source registers
  logic [0:127] src_reg_a, src_reg_b;
  // Immediate value, truncated based on instruction format
  logic [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  logic enable_reg_write;

  // Write Back Stage
  // Output value of Stage 3
  logic [0:127] wb_data;
  // Destination register for write back data
  logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  logic wb_enable_reg_write;
  // Indicates whether a branch is taken
  logic branch_is_taken;
  // Represents the delayed register address
  logic delayed_rt_addr;
  // Represents the delayed enable register write signal
  logic delayed_enable_reg_write;


  Byte dut (
     clock,
     reset,
```

```
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

// Set the initial state of the clock to zero
initial clock = 0;

// Toggle the clock value every 5 time units to simulate oscillation
always begin
  #5 clock = ~clock;
end

initial begin
  reset = 1;
  instr_format = 3'b000;
  // Set the opcode for the Count Ones in Bytes (cntb) operation
  op_code = 11'b01010110100;
  // Set the destination register address to $r3
  dest_reg_addr = 7'b0000011;
  // Set the value of source register A,  Halfwords: 16'h0010
  src_reg_a = 128'hABCDEF1234567890ABCDEF123456789;
  // Set the value of source register B, Halfwords: 16'h0001
  src_reg_b = 128'hFEDCBA0987654321FEDCBA098765432;
  imm_value = 0;
```

```
    enable_reg_write = 1;
    #6;
    // At 11ns, disable the reset, enabling the unit
    reset = 0;
    @(posedge clock);
    #4;
    // Set the opcode for the Average Bytes (avgb) operation
    op_code = 11'b00011010011;
    @(posedge clock);
    #4;
    // Set the opcode for the Absolute Differences of Bytes (absdb) operation
    op_code = 11'b00001010011;
    @(posedge clock);
    #4;
    // Set the opcode for the Sum Bytes into Halfwords (sumb) operation
    op_code = 11'b01001010011;
    @(posedge clock);
    #4;
    // Set the opcode for the No Operation (nop) instruction
    op_code = 0;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #100;
    op_code = 11'b00000000000;
    $stop;
  end
endmodule
```

Byte Waveform:



Local Store:

Local Store:

“Local Store” module is important for managing local memory operations in a processor design. This module helps with tasks such as loading and storing quadwords in local memory while also overseeing the write back stage, delayed register data, and enabling register writes based on instruction format and opcode. Through a variety of input singles including clock, reset, opcode, instruction format, register values, immediate values, and control flags, the module changes local memory operations according to the instructions received.

Upon receiving instruction and relevant data, the “Local Store” module executes different operations based on the format of the instructions and opcode. For example, it handles load and store quadword operations with different addressing modes, like x-form, d-form, and a-form. By calculating the memory addresses and accessing local memory appropriately, the module ensures the accurate retrieval or storage of quadword data. Additionally, it also manages the program counter to determine the memory addresses for instruction relative load and store operations. Through handling of memory operations and control signals, this module helps to guarantee the consistency of data transfers between the processor and the local memory. This module's functionality is important for maintaining efficient data processing and memory management within the overall architecture of the processor.

Local Store Code:

```
/*************************************************************************************************************
 * Module: Local Store
 * Author: Noah Merone
```

```
 *-----------------------------------------------------------------------------------------------------
 * Description:
 *    This module handles the local memory operations such as loading and storing quadwords. It manages the
 *    write-back stage, delayed register data, and enables register write based on the instruction format and
 *    opcode.
 *-----------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Decoded opcode truncated based on instr_format
 *   - instr_format: Format of the instruction, used with op_code and imm_value
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Value of source register A
 *   - src_reg_b: Value of source register B
 *   - store_reg: Value to be stored in memory
 *   - imm_value: Immediate value truncated based on instr_format
 *   - enable_reg_write: Flag indicating whether the current instruction writes to the register table
 *   - branch_is_taken: Signal indicating if a branch was taken
 *-----------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Output value of Stage 3
 *   - wb_reg_addr: Destination register for wb_data
 *   - wb_enable_reg_write: Will wb_data write to the register table
 *   - delayed_rt_addr: Destination register for wb_data, delayed by one clock cycle
 *   - delayed_enable_reg_write: Will wb_data write to the register table, delayed by one clock cycle
 *-----------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - delayed_rt_data: Staging register for calculated values, delayed by one clock cycle
 *   - local_mem: 32KB local memory for storing quadwords
 *   - program_counter: Current program counter value
 ******************************************************************************************************/

module Local_Store (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    store_reg,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

  input clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  input [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  input [2:0] instr_format;
  // Destination register address
  input [0:6] dest_reg_addr;
  // Values of source registers
```

```
input [0:127] src_reg_a, src_reg_b, store_reg;
// Immediate value, truncated based on instruction format
input [0:17] imm_value;
// Flag indicating if the current instruction will write to the Register Table
input enable_reg_write;
// Was branch taken?
input branch_is_taken;

// Write Back Stage
// Output value of Stage 3
output logic [0:127] wb_data;
// Destination register for write back data
output logic [0:6] wb_reg_addr;
// Flag indicating if the write back data will be written to the Register Table
output logic wb_enable_reg_write;

// Internal Signals
// Staging register for calculated values
logic [5:0][0:127] delayed_rt_data;
// Destination register for write back data
output logic [5:0][0:6] delayed_rt_addr;
// Flag indicating if the write back data will be written to the Register Table
output logic [5:0] delayed_enable_reg_write;
// 32KB local memory
logic [0:127] local_mem[0:2047];
// Program Counter
logic [31:0] program_counter;

always_comb begin
  wb_data = delayed_rt_data[4];
  wb_reg_addr = delayed_rt_addr[4];
  wb_enable_reg_write = delayed_enable_reg_write[4];
end

always_ff @(posedge clock) begin
  if (reset == 1) begin
    delayed_rt_data[5] <= 0;
    delayed_rt_addr[5] <= 0;
    delayed_enable_reg_write[5] <= 0;
    for (int i = 0; i < 5; i = i + 1) begin
      delayed_rt_data[i] <= 0;
      delayed_rt_addr[i] <= 0;
      delayed_enable_reg_write[i] <= 0;
    end
    for (logic [0:11] i = 0; i < 2048; i = i + 1) begin
      //mem[i] <= 0;
      local_mem[i] <= {i * 4, (i * 4 + 1), (i * 4 + 2), (i * 4 + 3)};
    end
  end else begin
    // Increment the program counter by 4, assuming each instruction is 4 bytes long
    program_counter <= program_counter + 4;
    delayed_rt_data[5] <= delayed_rt_data[4];
    delayed_rt_addr[5] <= delayed_rt_addr[4];
    delayed_enable_reg_write[5] <= delayed_enable_reg_write[4];
    for (int i = 0; i < 4; i = i + 1) begin
      delayed_rt_data[i+1] <= delayed_rt_data[i];
      delayed_rt_addr[i+1] <= delayed_rt_addr[i];
      delayed_enable_reg_write[i+1] <= delayed_enable_reg_write[i];
    end
    //nop : No Operation (Load)
    if (instr_format == 0 && op_code == 0) begin
```

```
  delayed_rt_data[0] <= 0;
  delayed_rt_addr[0] <= 0;
  delayed_enable_reg_write[0] <= 0;
 end else begin
  delayed_rt_addr[0] <= dest_reg_addr;
  delayed_enable_reg_write[0] <= enable_reg_write;
  // If a branch was taken in the last cycle, cancel the last instruction
  if (branch_is_taken) begin
   delayed_rt_data[0] <= 0;
   delayed_rt_addr[0] <= 0;
   delayed_enable_reg_write[0] <= 0;
  end else if (instr_format == 0) begin
   case (op_code)
    //lqx : Load Quadword (x-form)
    11'b00111000100: begin
     int addr_sum;
     addr_sum = $signed(src_reg_a[0:31]) + $signed(src_reg_b[0:31]);
     delayed_rt_data[0] <= local_mem[addr_sum];
    end
    //stqx : Store Quadword (x-form)
    11'b00101000100: begin
     int addr_sum;
     addr_sum = $signed(src_reg_a[0:31]) + $signed(src_reg_b[0:31]);
     local_mem[addr_sum] <= store_reg;
     delayed_enable_reg_write[0] <= 0;
    end
    default begin
     delayed_rt_data[0] <= 0;
     delayed_rt_addr[0] <= 0;
     delayed_enable_reg_write[0] <= 0;
    end
   endcase
  end else if (instr_format == 4) begin
   case (op_code[3:10])
    //lqd : Load Quadword (d-form)
    8'b00110100: begin
     int addr_sum;
     addr_sum = $signed(src_reg_a[0:31]) + $signed(imm_value[8:17]);
     delayed_rt_data[0] <= local_mem[addr_sum];
    end
    //stqd : Store Quadword (d-form)
    8'b00100100: begin
     int addr_sum;
     addr_sum = $signed(src_reg_a[0:31]) + $signed(imm_value[8:17]);
     local_mem[addr_sum] <= store_reg;
     delayed_enable_reg_write[0] <= 0;
    end
    default begin
     delayed_rt_data[0] <= 0;
     delayed_rt_addr[0] <= 0;
     delayed_enable_reg_write[0] <= 0;
    end
   endcase
  end else if (instr_format == 5) begin
   case (op_code[2:10])
    //lqa : Load Quadword (a-form)
    9'b001100001: begin
     int addr;
     addr = $signed(imm_value[2:17]);
     delayed_rt_data[0] <= local_mem[addr];
    end
```

```
        //stqa : Store Quadword (a-form)
        9'b001000001: begin
          int addr;
          addr = $signed(imm_value[2:17]);
          local_mem[addr] <= store_reg;
          delayed_enable_reg_write[0] <= 0;
        end
        //lqr: Load Quadword Instruction Relative (a-form)
        9'b001100111: begin
          //Calculate Load Store Address (LSA)
          reg [31:0] LSA;
          LSA = (({$signed(imm_value[15]), imm_value[2:15], 2'b00}) + program_counter) &
              32'hFFFFFFF0;
          //Load value from Local Memory using LSA
          delayed_rt_data[0] <= local_mem[LSA[31:2]];
        end
        //stqr: Store Quadword Instruction Relative (a-form)
        9'b001000111: begin
          //Calculate Load Store Address (LSA)
          reg [31:0] LSA;
          LSA = (({$signed(imm_value[15]), imm_value[2:15], 2'b00}) + program_counter) &
              32'hFFFFFFF0;
          //Store value to Local Memory using LSA
          local_mem[LSA[31:2]] <= store_reg;
          //No need to set delayed_enable_reg_write[0] as it's a store operation
        end
        default begin
          delayed_rt_data[0] <= 0;
          delayed_rt_addr[0] <= 0;
          delayed_enable_reg_write[0] <= 0;
        end
      endcase
    end else begin
      delayed_rt_data[0] <= 0;
      delayed_rt_addr[0] <= 0;
      delayed_enable_reg_write[0] <= 0;
    end
  end
 end
endmodule
```

Local Store Test Bench:

“Local Store” testbench module serves as a validation tool for verifying the functionality of the local store modules. Using simulation and monitoring the outputs, this testbench module helps to ensure that the local store module behaves as expected, handling various memory operations accurately and efficiently. It uses input singles including clock pulses, reset conditions, opcode, instruction formats, register values, immediate values, and control flags, to simulate different scenarios and validate the behavior of the local store module under different conditions.

This testbench module operates by simulating the execution of various memory operations, like loading and storing quad words, across multiple instruction formats and addressing modes. It configures the local

store module with specific operation codes and instruction format combinations. It also configures appropriate register values and immediate values to emulate real world scenarios which could be encountered during operation of the processor. By learning from these output signals, including the write-back data, destination register addresses, and enable register write flags, this testbench module verifies that the local store module successfully executes each memory operation and handles data transfers between the processor and local memory without any errors. With testing and debugging, the local store testbench module ensures reliability of the local store module.

Local Store Test Bench Code:

```
/*************************************************************************************************************
 * Module: Local Store Testbench
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *    This testbench module verifies the functionality of the Local Store module by providing stimulus and
 *    monitoring the outputs.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Decoded opcode truncated based on instr_format
 *   - instr_format: Format of the instruction, used with op_code and imm_value
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Value of source register A
 *   - src_reg_b: Value of source register B
 *   - store_reg: Value to be stored in memory
 *   - imm_value: Immediate value truncated based on instr_format
 *   - enable_reg_write: Flag indicating whether the current instruction writes to the register table
 *   - branch_is_taken: Signal indicating if a branch was taken
 *-------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Output value of Stage 3
 *   - wb_reg_addr: Destination register for wb_data
 *   - wb_enable_reg_write: Will wb_data write to the register table
```

```
*   - delayed_rt_addr: Destination register for wb_data, delayed by one clock cycle
*   - delayed_enable_reg_write: Will wb_data write to the register table, delayed by one clock cycle
*-----------------------------------------------------------------------------------------------------
* Internal Signals:
*   - delayed_rt_data: Staging register for calculated values, delayed by one clock cycle
*   - local_mem: 32KB local memory for storing quadwords
*   - program_counter: Current program counter value
*****************************************************************************************************/

module Local_Store_TB ();
  logic clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  logic [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  logic [ 2:0] instr_format;
  // Destination register address
  logic [ 0:6] dest_reg_addr;
  // Values of source registers
  logic [0:127] src_reg_a, src_reg_b, store_reg_odd;
  // Immediate value, truncated based on instruction format
  logic [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  logic enable_reg_write;

  // Write Back Stage
  // Output value of Stage 3
  logic [0:127] wb_data;
  // Destination register for write back data
  logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  logic wb_enable_reg_write;
  // Represents the store register signal
  logic store_reg;
```

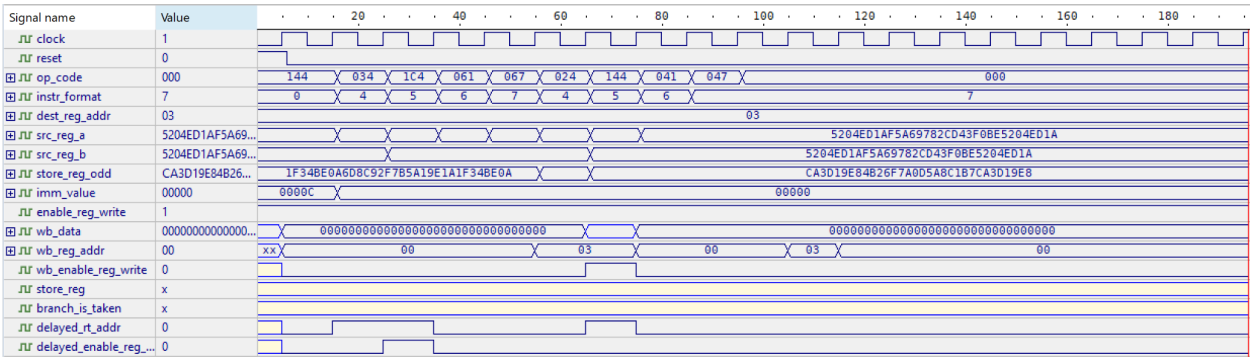
```
// Indicates whether a branch is taken
logic branch_is_taken;
// Represents the delayed register address
logic delayed_rt_addr;
// Represents the delayed enable register write signal
logic delayed_enable_reg_write;



Local_Store dut (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    store_reg,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

// Set the initial state of the clock to zero
initial clock = 0;

// Toggle the clock value every 5 time units to simulate oscillation
always begin
  #5 clock = ~clock;
end
```

```
initial begin
  reset = 1;
  instr_format = 3'b000;
  // Set the opcode for the Store Quadword (stqx) operation
  op_code = 11'b00101000100;
  // Set the destination register address to $r3
  dest_reg_addr = 7'b0000011;
  //Halfwords: 16'h0001
  src_reg_a = 128'hF9EBA5CD6078C31425F79B42F9EBA5CD;
  //Halfwords: 16'h0001
  src_reg_b = 128'hD5B91E846F7A3CE590D2E4D9D5B91E84;
  store_reg_odd = 128'h1F34BE0A6D8C92F7B5A19E1A1F34BE0A;
  imm_value = 12;
  enable_reg_write = 1;
  #6;
  // At 11ns, disable the reset, enabling the unit
  reset = 0;
  @(posedge clock);
  #1;
  // Load Quadword (d-form)
  op_code = 8'b00110100;
  instr_format = 4'b1100;
  dest_reg_addr = 7'b0000011;
  src_reg_a = 128'hC39A50EB8FD64B12A7EFC3BBC39A50EB;
  imm_value = 7'b0000000;
  @(posedge clock);
  #1;
  // Load Quadword (x-form)
  op_code = 11'b00111000100;
  instr_format = 4'b1101;
  dest_reg_addr = 7'b0000011;
  src_reg_a = 128'h7DC680A1EF59D438A1CDE2A27DC680A1;
  src_reg_b = 128'h9C86514EBDA07F53A1E25DC69C86514E;
  @(posedge clock);
  #1;
```

```
// Load Quadword (a-form)
op_code = 9'b001100001;
instr_format = 4'b1110;
dest_reg_addr = 7'b0000011;
src_reg_a = 128'h9C13F6DB0E24A85CB6F9A2F69C13F6DB;
imm_value = 7'b0000000;
@(posedge clock);
#1;
// Load Quadword Instruction Relative (a-form)
op_code = 9'b001100111;
instr_format = 4'b1111;
dest_reg_addr = 7'b0000011;
src_reg_a = 128'hF6D5803C29B6E8C14A3F01D7F6D5803C;
imm_value = 7'b0000000;
@(posedge clock);
#1;
// Store Quadword (d-form)
op_code = 8'b00100100;
instr_format = 4'b1100;
src_reg_a = 128'h86E2A70D5F1D48CB9F3E0A5A86E2A70D;
store_reg_odd = 128'h8F7E20BC64F5A1C3E9D0B73F8F7E20BC;
imm_value = 7'b0000000;
@(posedge clock);
#1;
// Store Quadword (x-form)
op_code = 11'b00101000100;
instr_format = 4'b1101;
src_reg_a = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
src_reg_b = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
store_reg_odd = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
@(posedge clock);
#1;
// Store Quadword (a-form)
op_code = 9'b001000001;
instr_format = 4'b1110;
```

```
    src_reg_a = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
    store_reg_odd = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
    imm_value = 7'b0000000;
    @(posedge clock);
    #1;
    // Store Quadword Instruction Relative (a-form)
    op_code = 9'b001000111;
    instr_format = 4'b1111;
    src_reg_a = 128'h5204ED1AF5A69782CD43F0BE5204ED1A;
    store_reg_odd = 128'hCA3D19E84B26F7A0D5A8C1B7CA3D19E8;
    imm_value = 7'b0000000;
    @(posedge clock);
    #1;
    // Set the opcode for the No Operation (nop) instruction
    op_code = 0;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #100;
    op_code = 11'b00000000000;
    $stop;
  end
endmodule
```

Local Store Waveform:



Permute:

Permute:

The “Permute” module is responsible for executing various operations based on the opcode and the instruction format inputs. It performs operations like shifting, rotating, and storing data, helping in manipulating data within the processor pipeline. Operating across register forwarding and write back stages, this module helps with the flow of data and control signals to ensure execution of instructions.

At the core, this module takes in inputs like clock signals, reset conditions, opcode, instruction formats, register values, immediate values, and control flags. It will then process these inputs to produce outputs that include write back data, destination register addresses, and enable register write flags. The module handles a broad range of instruction formats and opcode combinations, executing operations like shifting and rotating data.

Using a combination of combinational and sequential logic, the “Permute” module responds to clock pulses and resets to maintain the integrity of data flow in the processor pipeline. IT employs conditional statements and case structures to understand the appropriate operation to perform based on the provided opcodes and instructions. Through execution of instructions and handling of data manipulation tasks, this module helps with the performance of the operations and accuracy in the processor design.

Permute Code:

```
/*************************************************************************************************************
 * Module: Permute
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *     This module performs various operations based on the opcode and instruction format, such as shifting,
 *     rotating, and storing data. It handles both register forwarding and write-back stages.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Decoded opcode truncated based on instr_format
 *   - instr_format: Format of the instruction, used with op_code and imm_value
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Value of source register A
 *   - src_reg_b: Value of source register B
 *   - imm_value: Immediate value truncated based on instr_format
 *   - enable_reg_write: Flag indicating whether the current instruction writes to the register table
 *   - branch_is_taken: Signal indicating if a branch was taken
 *-------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Output value of Stage 3
```

```
 *   - wb_reg_addr: Destination register for wb_data
 *   - wb_enable_reg_write: Will wb_data write to the register table
 *-------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - delayed_rt_data: Staging register for calculated values
 *   - delayed_rt_addr: Destination register for wb_data, delayed by one clock cycle
 *   - delayed_enable_reg_write: Will wb_data write to the register table, delayed by one clock cycle
 ********************************************************************************************************/

module Permute (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

  input clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  input [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  input [2:0] instr_format;
  // Destination register address
  input [0:6] dest_reg_addr;
  // Values of source registers
  input [0:127] src_reg_a, src_reg_b;
  // Immediate value, truncated based on instruction format
  input [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  input enable_reg_write;
  // Was branch taken?
  input branch_is_taken;

  // Write Back Stage
  // Output value of Stage 3
  output logic [0:127] wb_data;
  // Destination register for write back data
  output logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic wb_enable_reg_write;

  // Internal Signals
  // Staging register for calculated values
  logic [3:0][0:127] delayed_rt_data;
  // Destination register for write back data
  output logic [3:0][0:6] delayed_rt_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic [3:0] delayed_enable_reg_write;
```

```
// 7-bit counter used for loops
logic [  6:0] i;
// Temporary variables used for intermediate calculations
logic [0:127] temporary_variable;

always_comb begin
  wb_data = delayed_rt_data[2];
  wb_reg_addr = delayed_rt_addr[2];
  wb_enable_reg_write = delayed_enable_reg_write[2];
end

always_ff @(posedge clock) begin
  if (reset == 1) begin
    delayed_rt_data[3] <= 0;
    delayed_rt_addr[3] <= 0;
    delayed_enable_reg_write[3] <= 0;
    for (i = 0; i < 3; i = i + 1) begin
      delayed_rt_data[i] <= 0;
      delayed_rt_addr[i] <= 0;
      delayed_enable_reg_write[i] <= 0;
    end
    temporary_variable <= 0;
  end else begin
    delayed_rt_data[3] <= delayed_rt_data[2];
    delayed_rt_addr[3] <= delayed_rt_addr[2];
    delayed_enable_reg_write[3] <= delayed_enable_reg_write[2];
    delayed_rt_data[2] <= delayed_rt_data[1];
    delayed_rt_addr[2] <= delayed_rt_addr[1];
    delayed_enable_reg_write[2] <= delayed_enable_reg_write[1];
    delayed_rt_data[1] <= delayed_rt_data[0];
    delayed_rt_addr[1] <= delayed_rt_addr[0];
    delayed_enable_reg_write[1] <= delayed_enable_reg_write[0];
    //nop : No Operation (Load)
    if (instr_format == 0 && op_code == 0) begin
      delayed_rt_data[0] <= 0;
      delayed_rt_addr[0] <= 0;
      delayed_enable_reg_write[0] <= 0;
    end else begin
      delayed_rt_addr[0] <= dest_reg_addr;
      delayed_enable_reg_write[0] <= enable_reg_write;
      if (branch_is_taken) begin
        delayed_rt_data[0] <= 0;
        delayed_rt_addr[0] <= 0;
        delayed_enable_reg_write[0] <= 0;
      end else if (instr_format == 0) begin
        case (op_code)
          //shlqbi : Shift Left Quadword by Bits
          11'b00111011011: begin
            int shift_amount;
            shift_amount = src_reg_b[29:31];
            delayed_rt_data[0] <= src_reg_a << shift_amount;
          end
          //shlqby rt, ra, rb : Shift Left Quadword by Bytes
          11'b00111011111: begin
            int shift_amount;
            shift_amount = src_reg_b[27:31] * 8;
            delayed_rt_data[0] <= src_reg_a << shift_amount;
          end
          //rotqby rt,ra,rb Rotate Quadword by Bytes
          11'b00111011100: begin
            temporary_variable = src_reg_b[28:31];
```

```
      for (int b = 0; b <= 15; b++) begin
       if (b + temporary_variable < 16) begin
        for (int i = b * 8; i < (b * 8 + 8); i++) begin
         delayed_rt_data[0][i] = src_reg_a[i+temporary_variable*8];
        end
       end else begin
        for (int i = b * 8; i < (b * 8 + 8); i++) begin
         delayed_rt_data[0][i] = src_reg_a[i+temporary_variable*8-16*8];
        end
       end
      end
     end
    endcase
  end else if (instr_format == 2) begin
    case (op_code)
     //shlqbi : Shift Left Quadword by Bits
     11'b00111011011: begin
      int shift_amount;
      int new_index;
      shift_amount = imm_value & 7'b0000111;
      // Initialize delayed_rt_data[0] to avoid uninitialized elements
      for (int i = 0; i < 128; i++) begin
       delayed_rt_data[0][i] = 0;
      end
      for (int b = 0; b < 128; b++) begin
       new_index = b + shift_amount;
       if (new_index < 128) begin
        delayed_rt_data[0][b] = src_reg_a[new_index];
       end
      end
     end
     //shlqbii rt, ra, value Shift Left Quadword by Bits Immediate
     11'b00111111011: begin
      int shift_amount;
      int new_index;
      shift_amount = imm_value & 7'b0001111;
      // Initialize delayed_rt_data[0] to avoid uninitialized elements
      for (int i = 0; i < 128; i++) begin
       delayed_rt_data[0][i] = 0;
      end
      for (int b = 0; b < 128; b = b + 1) begin
       new_index = b + shift_amount;
       if (new_index < 128) begin
        delayed_rt_data[0][b] = src_reg_a[new_index];
       end
      end
     end
     //shlqbyi rt,ra,value Shift Left Quadword by Bytes Immediate
     11'b00111111111: begin
      int shift_amount;
      shift_amount = imm_value & 7'b0001111;
      // Initialize delayed_rt_data[0] to avoid uninitialized elements
      for (int i = 0; i < 128; i++) begin
       delayed_rt_data[0][i] = 0;
      end
      for (int b = 0; b < 16; b++) begin
       if (b + shift_amount < 16) begin
        for (int i = 0; i < 8; i++) begin
         delayed_rt_data[0][b*8+i] = src_reg_a[(b+shift_amount)*8+i];
        end
       end
```

```
        end
      end
      //shlqbybi rt,ra,rb Shift Left Quadword by Bytes from Bit Shift Count
      11'b00111001111: begin
        int bit_shift_count;
        bit_shift_count = src_reg_b[24:28];
        // Initialize delayed_rt_data[0] to avoid uninitialized elements
        for (int i = 0; i < 128; i++) begin
          delayed_rt_data[0][i] = 8'h00;
        end
        for (int b = 0; b < 16; b = b + 1) begin
          if (b + bit_shift_count < 16) begin
            delayed_rt_data[0][b*8+:8] = src_reg_a[(b+bit_shift_count)*8+:8];
          end
        end
      end
      //rotqbyi rt, ra, imm7 Rotate Quadword by Bytes Immediate
      11'b00111111100: begin
        int rotation_amount;
        rotation_amount = imm_value & 7'b0001111;
        // Initialize delayed_rt_data[0] to avoid uninitialized elements
        for (int i = 0; i < 128; i++) begin
          delayed_rt_data[0][i] = 0;
        end
        for (int b = 0; b < 16; b++) begin
          if (b + rotation_amount < 16) begin
            for (int i = 0; i < 8; i++) begin
              delayed_rt_data[0][b*8+i] = src_reg_a[(b+rotation_amount)*8+i];
            end
          end else begin
            for (int i = 0; i < 8; i++) begin
              delayed_rt_data[0][b*8+i] = src_reg_a[(b+rotation_amount-16)*8+i];
            end
          end
        end
      end
      //rotqbybi rt,ra,rb Rotate Quadword by Bytes from Bit Shift Count
      11'b00111001100: begin
        int bit_shift_count;
        bit_shift_count = src_reg_b[24:28];
        // Initialize delayed_rt_data[0] to avoid uninitialized elements
        for (int i = 0; i < 128; i++) begin
          delayed_rt_data[0][i] = 0;
        end
        for (int b = 0; b < 16; b = b + 1) begin
          if (b + bit_shift_count < 16) begin
            delayed_rt_data[0][b*8+:8] = src_reg_a[(b+bit_shift_count)*8+:8];
          end else begin
            delayed_rt_data[0][b*8+:8] = src_reg_a[(b+bit_shift_count-16)*8+:8];
          end
        end
      end
      default begin
        delayed_rt_data[0] = 0;
        delayed_rt_addr[0] = 0;
        delayed_enable_reg_write[0] = 0;
      end
    endcase
  end
 end
end
```

```
  end
endmodule
```

Permute Test Bench:

The “Permute” testbench module serves as a simulation environment for verification of the “Permute” module. This provides insight into the generation of input signals and outputs to ensure that the module is performing as expected.

This testbench module includes inputs like clock signals, reset conditions, opcode, instruction formats, register values, immediate values, and control flags, mirroring the inputs required by the "Permute" module. This helps execution of test cases by setting the inputs to specific values and learning from the results in the outputs. By toggling the clock signal, initialing input values, and advancing time in steps, the testbench simulates the operation of the processor pipeline and various functionalities of the “Permute” module.

Through careful testing and cases, this testbench can validate the successfulness of operations performed by the “Permute” module, including shifting, rotating, and storing data. It also verifies that the module produces the expected outputs for a unique combination of inputs and outputs. In the end, the “Permute” testbench is important in contributing to the quality of the final design of the processor.

Permute Test Bench Code:

```
/*************************************************************************************************************
 * Module: Permute Testbench
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *    This module serves as the testbench for the Permute module. It provides stimulus to the Permute module
 *    by supplying input signals and monitors the outputs for correctness.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Decoded opcode truncated based on instr_format
 *   - instr_format: Format of the instruction, used with op_code and imm_value
 *   - dest_reg_addr: Destination register address
```

```
*   - src_reg_a: Value of source register A
*   - src_reg_b: Value of source register B
*   - imm_value: Immediate value truncated based on instr_format
*   - enable_reg_write: Flag indicating whether the current instruction writes to the register table
*   - branch_is_taken: Signal indicating if a branch was taken
*----------------------------------------------------------------------------------------------------
* Outputs:
*   - wb_data: Output value of Stage 3
*   - wb_reg_addr: Destination register for wb_data
*   - wb_enable_reg_write: Will wb_data write to the register table
*----------------------------------------------------------------------------------------------------
* Internal Signals:
*   - delayed_rt_data: Staging register for calculated values
*   - delayed_rt_addr: Destination register for wb_data, delayed by one clock cycle
*   - delayed_enable_reg_write: Will wb_data write to the register table, delayed by one clock cycle
****************************************************************************************************/

module Permute_TB ();
  logic clock, reset;

  // Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  logic [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  logic [ 2:0] instr_format;
  // Destination register address
  logic [ 0:6] dest_reg_addr;
  // Values of source registers
  logic [0:127] src_reg_a, src_reg_b;
  // Immediate value, truncated based on instruction format
  logic [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  logic enable_reg_write;

  // Write Back Stage
```

```
// Output value of Stage 3
logic [0:127] wb_data;
// Destination register for write back data
logic [0:6] wb_reg_addr;
// Flag indicating if the write back data will be written to the Register Table
logic wb_enable_reg_write;
// Indicates whether a branch is taken
logic branch_is_taken;
// Represents the delayed register address
logic delayed_rt_addr;
// Represents the delayed enable register write signal
logic delayed_enable_reg_write;



Permute dut (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

// Set the initial state of the clock to zero
initial clock = 0;
```

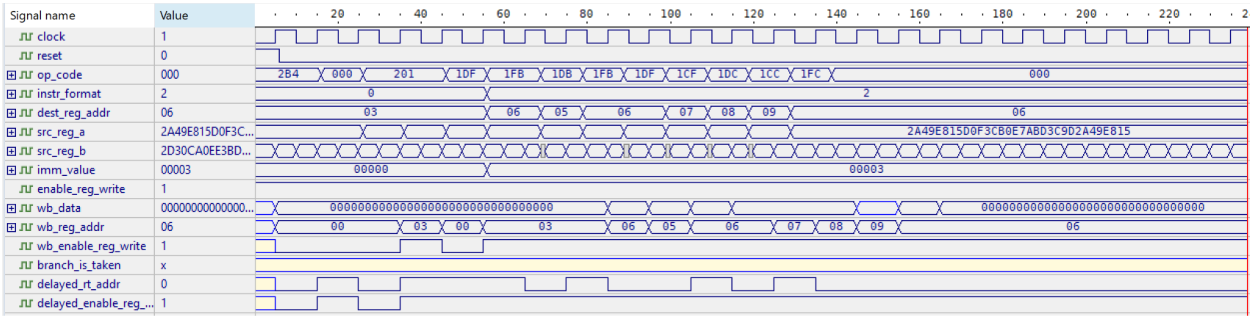
```
// Toggle the clock value every 5 time units to simulate oscillation
always begin
  #5 clock = ~clock;
  for (int i = 0; i < 8; i++) src_reg_b[i*16+:16] = src_reg_b[i*16+:16] + 1;
end

initial begin
  reset = 1;
  instr_format = 3'b000;
  // Set the opcode for the Shift Left Halfword (shlh) operation
  op_code = 11'b01010110100;
  // Set the destination register address to $r3
  dest_reg_addr = 7'b0000011;
  // Set the value of source register A, Halfwords: 16'h0010
  src_reg_a = 128'h5A7F38E2B0C4D69E18F2C86B5A7F38E2;
  // Set the value of source register B, Halfwords: 16'h0001
  src_reg_b = 128'h1E9D83CAB4267F0E2B8A64C11E9D83CA;
  imm_value = 0;
  enable_reg_write = 1;
  #6;
  // At 11ns, disable the reset, enabling the unit
  reset = 0;
  @(posedge clock);
  #1;
  // Set the opcode for the No Operation (nop) instruction
  op_code = 0;
  @(posedge clock);
  #1;
  op_code   = 11'b01000000001;
  src_reg_a = 128'hC7063E4F1B28D5A3F9A87F41C7063E4F;
  @(posedge clock);
  #1;
  src_reg_a = 128'hEF4528AB1D3976F825B6D843EF4528AB;
  @(posedge clock);
  #1;
```

```
op_code   = 11'b00111011111;
src_reg_a = 128'h1A73BF5D68294E1BCA053B921A73BF5D;
@(posedge clock);
#1;
// Set the opcode for the "Shift Left Quadword by Bits Immediate" operation (shlqbii rt, src_reg_a, value)
op_code = 11'b00111111011;
// Set the destination register address to $r6
dest_reg_addr = 7'b0000110;
// Set the instruction format to RI7-type
instr_format = 2;
// Set the immediate value to 3
imm_value = 7'b0000011;
src_reg_a = 128'hB7D84231F6A905E378CD720EB7D84231;
@(posedge clock);
#4;
// Testcase for Shift Left Quadword by Bits (shlqbi rt, src_reg_a, src_reg_b)
src_reg_b = 128'h04987A6DB123FEDCFAE9080D04987A6D;
src_reg_a = 128'h96FC4D1E82AB0E64F3D89B2A96FC4D1E;
op_code = 11'b00111011011;
dest_reg_addr = 7'b0000101;
@(posedge clock);
#4;
// Set the opcode for the "Shift Left Quadword by Bits Immediate" operation (shlqbii rt, src_reg_a, value)
op_code = 11'b00111111011;
// Set the destination register address to $r6
dest_reg_addr = 7'b0000110;
// Set the instruction format to RI7-type
instr_format = 2;
// Set the immediate value to 3
imm_value = 7'b0000011;
src_reg_a = 128'h3E50187ABF6C9D24E6A3B59B3E50187A;
@(posedge clock);
#4;
// Testcase for Shift Left Quadword by Bytes (shlqby rt, src_reg_a, src_reg_b)
src_reg_b = 128'h8D7E20BC64F5A1C3E9D0B73F8D7E20BC;
```

```
src_reg_a = 128'hFAC825D7B39461E20E37A7B5FAC825D7;
op_code = 11'b00111011111;
dest_reg_addr = 7'b0000110;
@(posedge clock);
#4;
// Testcase for Shift Left Quadword by Bytes from Bit Shift Count (shlqbybi rt, src_reg_a, src_reg_b)
src_reg_b = 128'h0E6D9CAB724F158D37A109380E6D9CAB;
src_reg_a = 128'h29C673E10A4FEDB21B359A5629C673E1;
op_code = 11'b00111001111;
dest_reg_addr = 7'b0000111;
@(posedge clock);
#4;
// Testcase for Rotate Quadword by Bytes (rotqby rt, src_reg_a, src_reg_b)
src_reg_b = 128'hB804ED1AF5A69782CD43F0BEB804ED1A;
src_reg_a = 128'hA6E9D8013FCB24A857D0ECA3A6E9D801;
op_code = 11'b00111011100;
dest_reg_addr = 7'b0001000;
@(posedge clock);
#4;
// Testcase for Rotate Quadword by Bytes from Bit Shift Count (rotqbybi rt, src_reg_a, src_reg_b)
src_reg_b = 128'h2D18C9F6E3A5B470F1E862492D18C9F6;
src_reg_a = 128'hBEAF572C4D1E0F38C62B91E6BEAF572C;
op_code = 11'b00111001100;
dest_reg_addr = 7'b0001001;
@(posedge clock);
#4;
// Set the opcode for the "Rotate Quadword by Bytes Immediate" operation (rotqbyi rt, src_reg_a, imm7)
op_code = 11'b00111111100;
// Set the destination register address to $r6
dest_reg_addr = 7'b0000110;
// Set the instruction format to RI7-type
instr_format = 2;
// Set the immediate value to 3
imm_value = 7'b0000011;
src_reg_a = 128'h2A49E815D0F3CB0E7ABD3C9D2A49E815;
```

```
    @(posedge clock);
    #4;
    op_code = 0;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #100;
    op_code = 11'b00000000000;
    $stop;
  end
endmodule
```

Permute Waveform:



Simple Fixed 1:

Simple Fixed 1:

“Simple fixed 1’ module serves as a fixed point arithmetic unit. This module is designed to perform different arithmetic operations and logical operations based on the given instructions and inputs. This module consists of separate stages for register file/ forwarding and write back.

In the register file/ forwarding stage, the inputs include clock, reset, operation code (op_code), instruction format (instr_format), destination register address (dest_reg_addr), source register A (src_reg_a), source register B (src_reg_b), store register (store_reg), immediate value (imm_value), flag indicating register write operation (enable_reg_write), and a flag indicating whether a branch is taken (branch_is_taken). The outputs form this stage are the data to be written back to the register file (wb_data), the address of the

register to be written back (wb_reg_addr), and a flag indicating register write operation in the write back stage (wb_enable_reg_write).

The module's functionality is handling different instruction formats such as no operation (nop), arithmetic operations like addition and subtraction, logical operations like AND, OR, and XOR, as well as comparison operations like comparing equal, greater than, and logical greater than. Additionally, it supports immediate operations for loading immediate values into registers and performing arithmetic operations and logical operations with immediate values. The module also handles branching operations and has support for extended arithmetic operations. The functionality is successfully implemented through a combination of conditional statements and loop structures, ensuring efficient execution of instructions and proper handling of data.

Simple Fixed 1 Code:

```
/*************************************************************************************************************
 * Module: Simple Fixed 1
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *    This module implements a simple fixed-point arithmetic unit. It performs various arithmetic and logical
 *    operations based on the given instructions and inputs. The module contains separate stages for register
 *    file/forwarding and write back.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Operation code
 *   - instr_format: Instruction format
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Source register A
 *   - src_reg_b: Source register B
 *   - store_reg: Store register
 *   - imm_value: Immediate value
 *   - enable_reg_write: Flag indicating register write operation
 *   - branch_is_taken: Flag indicating branch taken
 *-------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Data to be written back to the register file
 *   - wb_reg_addr: Address of the register to be written back
 *   - wb_enable_reg_write: Flag indicating register write operation in WB stage
 *-------------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - delayed_rt_data: Delayed register data for forwarding
 *   - delayed_rt_addr: Delayed register address for forwarding
 *   - delayed_enable_reg_write: Delayed register write flag for forwarding
 *************************************************************************************************************/

module Simple_Fixed_1 (
   clock,
   reset,
   op_code,
   instr_format,
   dest_reg_addr,
```

```
    src_reg_a,
    src_reg_b,
    store_reg,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

  input clock, reset;

  //Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  input [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  input [2:0] instr_format;
  // Destination register address
  input [0:6] dest_reg_addr;
  // Values of source registers
  input [0:127] src_reg_a, src_reg_b, store_reg;
  // Immediate value, truncated based on instruction format
  input [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  input enable_reg_write;
  // Flag indicating if a branch was taken
  input branch_is_taken;

  //Write Back Stage
  // Output value of Stage 3
  output logic [0:127] wb_data;
  // Destination register for write back data
  output logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic wb_enable_reg_write;

  //Internal Signals
  // Staging register for calculated values
  logic [3:0][0:127] delayed_rt_data;
  // Destination register for write back data
  output logic [1:0][0:6] delayed_rt_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic [1:0] delayed_enable_reg_write;

  // 7-bit counter used for loops
  logic [6:0] i;

  // Define the maximum 32-bit signed value
  logic signed [31:0] max_value_32 = 32'h7FFFFFFF;
  // Define the minimum 32-bit signed value
  logic signed [31:0] min_value_32 = 32'h80000000;
  // Define the maximum 16-bit signed value
  logic signed [15:0] max_value_16 = 16'h7FFF;
  // Define the minimum 16-bit signed value
  logic signed [15:0] min_value_16 = 16'h8000;

  // A temporary variable used for intermediate computations
  logic [0:128] temporary_variable;
```

```
always_comb begin
  wb_data = delayed_rt_data[0];
  wb_reg_addr = delayed_rt_addr[0];
  wb_enable_reg_write = delayed_enable_reg_write[0];
end

always_ff @(posedge clock) begin
  if (reset == 1) begin
    delayed_rt_data[1] <= 0;
    delayed_rt_addr[1] <= 0;
    delayed_enable_reg_write[1] <= 0;
    delayed_rt_data[0] <= 0;
    delayed_rt_addr[0] <= 0;
    delayed_enable_reg_write[0] <= 0;
    temporary_variable = 0;
  end else begin
    delayed_rt_data[1] <= delayed_rt_data[0];
    delayed_rt_addr[1] <= delayed_rt_addr[0];
    delayed_enable_reg_write[1] <= delayed_enable_reg_write[0];

    //nop : No Operation (Execute)
    if (instr_format == 0 && op_code == 0) begin
      delayed_rt_data[0] <= 0;
      delayed_rt_addr[0] <= 0;
      delayed_enable_reg_write[0] <= 0;
    end else begin
      delayed_rt_addr[0] <= dest_reg_addr;
      delayed_enable_reg_write[0] <= enable_reg_write;
      if (branch_is_taken) begin
        delayed_rt_data[0] = 0;
        delayed_rt_addr[0] = 0;
        delayed_enable_reg_write[0] = 0;
      end else if (instr_format == 0) begin
        case (op_code)
          // cg rt, ra, rb Carry Generate
          11'b00011000010: begin
            for (int i = 0; i < 16; i = i + 4) begin
              // Declare local variables
              bit [31:0] a_segment;
              bit [31:0] b_segment;
              bit [32:0] sum;
              // Extract 32-bit segments from src_reg_a and src_reg_b
              a_segment = src_reg_a[(i*32)+:32];
              b_segment = src_reg_b[(i*32)+:32];
              // Add src_reg_a and src_reg_b with carry-in of 0
              sum = {1'b0, a_segment} + {1'b0, b_segment};
              // Assign the result to delayed_rt_data[0]
              delayed_rt_data[0][(i*32)+:32] = sum[31:0];
              // Set carry flag based on comparison of src_reg_a and src_reg_b
              delayed_rt_data[0][((i+1)*32)] = (b_segment >= a_segment) ? 1'b1 : 1'b0;
            end
          end
          // Borrow Generate
          11'b00001000010: begin
            for (int i = 0; i < 16; i = i + 4) begin
              // Declare local variables
              bit [31:0] a_segment;
              bit [31:0] b_segment;
              bit [32:0] difference;
              // Extract 32-bit segments from src_reg_a and src_reg_b
```

```
    a_segment = src_reg_a[(i*32)+:32];
    b_segment = src_reg_b[(i*32)+:32];
    // Subtract src_reg_b from src_reg_a with borrow-in of 0
    difference = {1'b0, a_segment} - {1'b0, b_segment};
    // Assign the result to delayed_rt_data[0]
    delayed_rt_data[0][(i*32)+:32] = difference[31:0];
    // Set borrow flag based on comparison of src_reg_a and src_reg_b
    delayed_rt_data[0][((i+1)*32)] = (b_segment > a_segment) ? 1'b1 : 1'b0;
  end
 end
 //ah : Add Halfword
 11'b00011001000: begin
  for (int i = 0; i < 16; i = i + 1) begin
    // Declare the sum variable without initialization
    int sum;
    // Calculate the sum of src_reg_a and src_reg_b
    sum = $signed(src_reg_a[(i*16)+:16]) + $signed(src_reg_b[(i*16)+:16]);
    // Check for overflow and underflow conditions
    if (sum >= max_value_16) delayed_rt_data[0][(i*16)+:16] = max_value_16;
    else if (sum <= min_value_16) delayed_rt_data[0][(i*16)+:16] = min_value_16;
    else delayed_rt_data[0][(i*16)+:16] = sum;
  end
 end
 //ah : Add Word
 11'b00011000000: begin
  for (int i = 0; i < 32; i = i + 1) begin
    // Declare the sum variable without initialization
    int sum;
    // Calculate the sum of src_reg_a and src_reg_b
    sum = $signed(src_reg_a[(i*32)+:32]) + $signed(src_reg_b[(i*32)+:32]);
    // Check for overflow and underflow conditions
    if (sum >= max_value_32) delayed_rt_data[0][(i*32)+:32] = max_value_32;
    else if (sum <= min_value_32) delayed_rt_data[0][(i*32)+:32] = min_value_32;
    else delayed_rt_data[0][(i*32)+:32] = sum;
  end
 end
 //sfh rt, ra, rb : Subtract from Halfword
 11'b00001001000: begin
  for (int i = 0; i < 16; i = i + 1) begin
    // Declare the difference variable without initialization
    int difference;
    // Calculate the difference between src_reg_b and src_reg_a
    difference = $signed(src_reg_b[(i*16)+:16]) - $signed(src_reg_a[(i*16)+:16]);
    // Check for overflow and underflow conditions
    if (difference >= max_value_16) delayed_rt_data[0][(i*16)+:16] = max_value_16;
    else if (difference <= min_value_16) delayed_rt_data[0][(i*16)+:16] = min_value_16;
    else delayed_rt_data[0][(i*16)+:16] = difference;
  end
 end
 //sf rt, ra, rb : Subtract from Word
 11'b00001000000: begin
  for (int i = 0; i < 32; i = i + 1) begin
    // Declare the difference variable without initialization
    int difference;
    // Calculate the difference between src_reg_b and src_reg_a
    difference = $signed(src_reg_b[(i*32)+:32]) - $signed(src_reg_a[(i*32)+:32]);
    // Check for overflow and underflow conditions
    if (difference >= max_value_32) delayed_rt_data[0][(i*32)+:32] = max_value_32;
    else if (difference <= min_value_32) delayed_rt_data[0][(i*32)+:32] = min_value_32;
    else delayed_rt_data[0][(i*32)+:32] = difference;
  end
```

```
end
// and
11'b00011000001: begin
  delayed_rt_data[0] = src_reg_a & src_reg_b;
end
// or
11'b00001000001: begin
  delayed_rt_data[0] = src_reg_a | src_reg_b;
end
// xor
11'b01001000001: begin
  delayed_rt_data[0] = src_reg_a ^ src_reg_b;
end
// nand
11'b00011001001: begin
  delayed_rt_data[0] = ~(src_reg_a & src_reg_b);
end
// ceqh rt, ra, rb Compare Equal Halfword
11'b01111001000: begin
  for (int i = 0; i < 16; i = i + 2) begin
    // Check if the halfwords at position i in src_reg_a and src_reg_b are equal
    if (src_reg_a[(i*8)+:16] == src_reg_b[(i*8)+:16]) begin
      // Set the result to 16'hFFFF if equal
      delayed_rt_data[0][(i*8)+:16] = 16'hFFFF;
    end else begin
      // Set the result to 16'h0000 if not equal
      delayed_rt_data[0][(i*8)+:16] = 16'h0000;
    end
  end
end
// ceq rt, ra, rb Compare Equal Word
11'b01111000000: begin
  for (int i = 0; i < 16; i = i + 4) begin
    // Check if the words at position i in src_reg_a and src_reg_b are equal
    if (src_reg_a[(i*8)+:32] == src_reg_b[(i*8)+:32]) begin
      // Set the result to 32'hFFFFFFFF if equal
      delayed_rt_data[0][(i*8)+:32] = 32'hFFFFFFFF;
    end else begin
      // Set the result to 32'h00000000 if not equal
      delayed_rt_data[0][(i*8)+:32] = 32'h00000000;
    end
  end
end
// cgth rt, ra, rb Compare Greater Than Halfword
11'b01001001000: begin
  for (int i = 0; i < 16; i = i + 2) begin
    // Check if the halfwords at position i in src_reg_a is greater than src_reg_b
    if ($signed(src_reg_a[(i*8)+:16]) > $signed(src_reg_b[(i*8)+:16])) begin
      // Set the result to 16'hFFFF if greater than
      delayed_rt_data[0][(i*8)+:16] = 16'hFFFF;
    end else begin
      // Set the result to 16'h0000 if not greater than
      delayed_rt_data[0][(i*8)+:16] = 16'h0000;
    end
  end
end
// cgt rt, ra, rb Compare Greater Than Word
11'b01001000000: begin
  for (int i = 0; i < 16; i = i + 4) begin
    // Compare each word in src_reg_a and src_reg_b
    if ($signed(src_reg_a[(i*8)+:32]) > $signed(src_reg_b[(i*8)+:32])) begin
```

```
          // Set to 32'hFFFFFFFF if src_reg_a > src_reg_b
          delayed_rt_data[0][(i*8)+:32] = 32'hFFFFFFFF;
        end else begin
          // Set to 32'h00000000 if src_reg_a <= src_reg_b
          delayed_rt_data[0][(i*8)+:32] = 32'h00000000;
        end
      end
    end
    // clgtb rt, ra, rb Compare Logical Greater Than Byte
    11'b01011010000: begin
      for (int i = 0; i < 16; i = i + 1) begin
        // Compare each byte in src_reg_a and src_reg_b
        if (src_reg_a[(i*8)+:8] > src_reg_b[(i*8)+:8]) begin
          // Set to 8'hFF if src_reg_a > src_reg_b
          delayed_rt_data[0][(i*8)+:8] = 8'hFF;
        end else begin
          // Set to 8'h00 if src_reg_a <= src_reg_b
          delayed_rt_data[0][(i*8)+:8] = 8'h00;
        end
      end
    end
    // clgth rt, ra, rb Compare Logical Greater Than Halfword
    11'b01011001000: begin
      for (int i = 0; i < 16; i = i + 2) begin
        // Compare each halfword in src_reg_a and src_reg_b
        if (src_reg_a[(i*8)+:16] > src_reg_b[(i*8)+:16]) begin
          // Set to 16'hFFFF if src_reg_a > src_reg_b
          delayed_rt_data[0][(i*8)+:16] = 16'hFFFF;
        end else begin
          // Set to 16'h0000 if src_reg_a <= src_reg_b
          delayed_rt_data[0][(i*8)+:16] = 16'h0000;
        end
      end
    end
    // clgt rt, ra, rb Compare Logical Greater Than Word
    11'b01011000000: begin
      for (int i = 0; i < 16; i = i + 4) begin
        // Compare each word in src_reg_a and src_reg_b
        if (src_reg_a[(i*8)+:32] > src_reg_b[(i*8)+:32]) begin
          // Set to 32'hFFFFFFFF if src_reg_a > src_reg_b
          delayed_rt_data[0][(i*8)+:32] = 32'hFFFFFFFF;
        end else begin
          // Set to 32'h00000000 if src_reg_a <= src_reg_b
          delayed_rt_data[0][(i*8)+:32] = 32'h00000000;
        end
      end
    end
    default begin
      delayed_rt_data[0] <= 0;
      delayed_rt_addr[0] <= 0;
      delayed_enable_reg_write[0] <= 0;
    end
  endcase
end else if (instr_format == 4) begin  //RI10-type
  case (op_code)
    //ahi rt, ra, imm10 : Add Halfword Immediate
    8'b00011101: begin
      for (int i = 0; i < 16; i = i + 1) begin
        // Calculate the sum of src_reg_a and the immediate value
        int sum;
        sum = $signed(src_reg_a[(i*16)+:16]) + $signed(imm_value[8:17]);
```

```
    // Clamp the sum within the specified range
    if (sum >= max_value_16) begin
      delayed_rt_data[0][(i*16)+:16] = max_value_16;
    end else if (sum <= min_value_16) begin
      delayed_rt_data[0][(i*16)+:16] = min_value_16;
    end else begin
      delayed_rt_data[0][(i*16)+:16] = sum;
    end
  end
end
//ai rt, ra, imm10 : Add Word Immediate
8'b00011100: begin
  for (int i = 0; i < 4; i = i + 1) begin
    // Calculate the sum of src_reg_a and the immediate value
    int sum;
    sum = $signed(src_reg_a[(i*32)+:32]) + $signed(imm_value[8:17]);
    // Clamp the sum within the specified range
    if (sum >= max_value_32) begin
      delayed_rt_data[0][(i*32)+:32] = max_value_32;
    end else if (sum <= min_value_32) begin
      delayed_rt_data[0][(i*32)+:32] = min_value_32;
    end else begin
      delayed_rt_data[0][(i*32)+:32] = sum;
    end
  end
end
//sfhi rt, ra, imm10 : Subtract from Halfword Immediate
8'b00001101: begin
  for (int i = 0; i < 16; i = i + 1) begin
    // Calculate the difference between the immediate value and src_reg_a
    int difference;
    difference = $signed(imm_value[8:17]) - $signed(src_reg_a[(i*16)+:16]);
    // Clamp the difference within the specified range
    if (difference >= max_value_16) begin
      delayed_rt_data[0][(i*16)+:16] = max_value_16;
    end else if (difference <= min_value_16) begin
      delayed_rt_data[0][(i*16)+:16] = min_value_16;
    end else begin
      delayed_rt_data[0][(i*16)+:16] = difference;
    end
  end
end
//sfi rt, ra, imm10 : Subtract from Word Immediate
8'b00001100: begin
  for (int i = 0; i < 32; i = i + 1) begin
    // Calculate the difference between the immediate value and src_reg_a
    int difference;
    difference = $signed(imm_value[8:17]) - $signed(src_reg_a[(i*32)+:32]);
    // Clamp the difference within the specified range
    if (difference >= max_value_32) begin
      delayed_rt_data[0][(i*32)+:32] = max_value_32;
    end else if (difference <= min_value_32) begin
      delayed_rt_data[0][(i*32)+:32] = min_value_32;
    end else begin
      delayed_rt_data[0][(i*32)+:32] = difference;
    end
  end
end
// ceqhi rt, ra, imm10 Compare Equal Halfword Immediate
8'b01111101: begin
  // Extract the immediate value
```

```
  int imm_halfword;
  imm_halfword = {imm_value[8], imm_value[8:17]};
  // Iterate over each halfword in src_reg_a
  for (int i = 0; i < 16; i = i + 2) begin
    // Compare with the immediate value
    if (src_reg_a[(i*8)+:16] == imm_halfword) begin
      delayed_rt_data[0][(i*8)+:16] = 16'hFFFF;
    end else begin
      delayed_rt_data[0][(i*8)+:16] = 16'h0000;
    end
  end
end
// ceqi rt, ra, imm10 Compare Equal Word Immediate
8'b01111100: begin
  // Extract the immediate value
  int imm_word;
  imm_word = {imm_value[8], imm_value[8:17]};
  // Iterate over each word in src_reg_a
  for (int i = 0; i < 16; i = i + 4) begin
    // Compare with the immediate value
    if (src_reg_a[(i*8)+:32] == imm_word) begin
      delayed_rt_data[0][(i*8)+:32] = 32'hFFFFFFFF;
    end else begin
      delayed_rt_data[0][(i*8)+:32] = 32'h00000000;
    end
  end
end
// cgthi rt, ra, imm10 Compare Greater Than Halfword Immediate
8'b01001101: begin
  // Extract the immediate value
  int imm_halfword;
  imm_halfword = {imm_value[8], imm_value[8:17]};
  // Iterate over each halfword in src_reg_a
  for (int i = 0; i < 16; i = i + 2) begin
    // Compare with the immediate value
    if ($signed(src_reg_a[(i*8)+:16]) > $signed(imm_halfword)) begin
      delayed_rt_data[0][(i*8)+:16] = 16'hFFFF;
    end else begin
      delayed_rt_data[0][(i*8)+:16] = 16'h0000;
    end
  end
end
// cgti rt, ra, imm10 Compare Greater Than Word Immediate
8'b01001100: begin
  // Extract the immediate value
  int imm_word;
  imm_word = {imm_value[8], imm_value[8:17]};
  // Iterate over each word in src_reg_a
  for (int i = 0; i < 16; i = i + 4) begin
    // Compare with the immediate value
    if ($signed(src_reg_a[(i*8)+:32]) > $signed(imm_word)) begin
      delayed_rt_data[0][(i*8)+:32] = 32'hFFFFFFFF;
    end else begin
      delayed_rt_data[0][(i*8)+:32] = 32'h00000000;
    end
  end
end
// clgtbi rt, ra, imm10 Compare Logical Greater Than Byte Immediate
8'b01011110: begin
  // Extract the immediate value
  int imm_byte;
```

```
      imm_byte = imm_value[10:17];
      // Iterate over each byte in src_reg_a
      for (int i = 0; i < 16; i = i + 1) begin
        // Compare with the immediate value
        if ($unsigned(src_reg_a[(i*8)+:8]) > $unsigned(imm_byte)) begin
          delayed_rt_data[0][(i*8)+:8] = 8'hFF;
        end else begin
          delayed_rt_data[0][(i*8)+:8] = 8'h00;
        end
      end
    end
    // clgthi rt, ra, imm10 Compare Logical Greater Than Halfword Immediate
    8'b01011101: begin
      // Extract the immediate value
      shortint imm_halfword;
      imm_halfword = {imm_value[8], imm_value[8:17]};
      // Iterate over each halfword in src_reg_a
      for (int i = 0; i < 16; i = i + 2) begin
        // Compare with the immediate halfword value
        if ($unsigned(src_reg_a[(i*8)+:16]) > $unsigned(imm_halfword)) begin
          delayed_rt_data[0][(i*8)+:16] = 16'hFFFF;
        end else begin
          delayed_rt_data[0][(i*8)+:16] = 16'h0000;
        end
      end
    end
    // clgti rt, ra, imm10 Compare Logical Greater Than Word Immediate
    8'b01011100: begin
      // Extract the immediate value
      bit [31:0] imm_word;
      imm_word = imm_value[8:17];
      // Iterate over each word in src_reg_a
      for (int i = 0; i < 16; i = i + 4) begin
        // Compare with the immediate word value
        if ($unsigned(src_reg_a[(i*8)+:32]) > $unsigned(imm_word)) begin
          delayed_rt_data[0][(i*8)+:32] = 32'hFFFFFFFF;
        end else begin
          delayed_rt_data[0][(i*8)+:32] = 32'h00000000;
        end
      end
    end
    default begin
      delayed_rt_data[0] = 0;
      delayed_rt_addr[0] = 0;
      delayed_enable_reg_write[0] = 0;
      temporary_variable = 0;
    end
  endcase
end else if (instr_format == 5) begin
  case (op_code)
    // ilh rt, imm16 Immediate Load Halfword
    9'b010000011: begin
      // Extract the immediate value
      bit [15:0] imm_halfword;
      imm_halfword = imm_value[2:17];
      // Iterate over each halfword in delayed_rt_data
      for (int i = 0; i < 16; i = i + 2) begin
        // Load the immediate halfword value
        delayed_rt_data[0][(i*8)+:16] = imm_halfword;
      end
    end
```

```
    // ilhu rt, imm16 Immediate Load Halfword Upper
    9'b010000010: begin
      // Extract the immediate value
      bit [15:0] imm_halfword_upper;
      imm_halfword_upper = imm_value[2:17];
      // Iterate over each word in delayed_rt_data
      for (int i = 0; i < 16; i = i + 4) begin
        // Load the immediate halfword upper value with zero-extended lower halfword
        delayed_rt_data[0][(i*8)+:32] = {imm_halfword_upper, 16'h0000};
      end
    end
    // il rt, imm16 Immediate Load Word
    9'b010000001: begin
      // Extract the immediate value
      bit [15:0] imm_word;
      imm_word = imm_value[2:17];
      // Iterate over each word in delayed_rt_data
      for (int i = 0; i < 4; i = i + 1) begin
        // Load the immediate word value into delayed_rt_data
        delayed_rt_data[0][(i*32)+:32] = $signed(imm_word);
      end
    end
    // iohl rt, imm16 Immediate Or Halfword Lower
    9'b011000001: begin
      // Extract the immediate value
      bit [15:0] imm_halfword;
      imm_halfword = imm_value[2:17];
      // Iterate over each word in delayed_rt_data
      for (int i = 0; i < 16; i = i + 4) begin
        // Load the immediate halfword value into the lower half of each word in delayed_rt_data
        delayed_rt_data[0][(i*8)+:16] = store_reg[(i*8)+:16] | imm_halfword;
        // Copy the upper half from store_reg
        delayed_rt_data[0][((i+2)*8)+:16] = store_reg[((i+2)*8)+:16];
      end
    end
    default begin
      delayed_rt_data[0] = 0;
      delayed_rt_addr[0] = 0;
      delayed_enable_reg_write[0] = 0;
      temporary_variable = 0;
    end
  endcase
end else if (instr_format == 6) begin
  case (op_code)
    // ila rt, imm18 Immediate Load Address
    7'b0100001: begin
      // Extract the immediate value
      bit [17:0] imm_address;
      imm_address = imm_value[0:17];
      // Iterate over each word in delayed_rt_data
      for (int i = 0; i < 16; i = i + 4) begin
        // Load the immediate address value into the lower half of each word in delayed_rt_data
        delayed_rt_data[0][(i*8)+:18] = imm_address;
        // Copy the upper half from store_reg
        delayed_rt_data[0][((i+2)*8)+:14] = store_reg[((i+2)*8)+:14];
      end
    end
    // addx rt, ra, rb Add Extended
    11'b01101000000: begin
      // Iterate over each word in delayed_rt_data
      for (int i = 0; i < 4; i = i + 1) begin
```

```
            // Calculate the sum with extension
            bit signed_sum;
            signed_sum = $signed(src_reg_a[(i*32)+:32]) + $signed(src_reg_b[(i*32)+:32]) +
                $signed(store_reg[(i*32)+:32]);
            // Check for overflow
            if (signed_sum >= max_value_32) begin
              delayed_rt_data[0][(i*32)+:32] = max_value_32;
            end else if (signed_sum <= min_value_32) begin
              delayed_rt_data[0][(i*32)+:32] = min_value_32;
            end else begin
              delayed_rt_data[0][(i*32)+:32] = signed_sum;
            end
          end
        end
        // sfx rt, ra, rb Subtract from Extended
        11'b01101000001: begin
          // Iterate over each word in delayed_rt_data
          for (int i = 0; i < 4; i = i + 1) begin
            // Calculate the difference with extension
            bit signed_diff;
            signed_diff = $signed(store_reg[(i*32)+:32]) - $signed(src_reg_b[(i*32)+:32]) -
                $signed(src_reg_a[(i*32)+:32]);
            // Check for overflow
            if (signed_diff >= max_value_32) begin
              delayed_rt_data[0][(i*32)+:32] = max_value_32;
            end else if (signed_diff <= min_value_32) begin
              delayed_rt_data[0][(i*32)+:32] = min_value_32;
            end else begin
              delayed_rt_data[0][(i*32)+:32] = signed_diff;
            end
          end
        end
        default begin
          delayed_rt_data[0] = 0;
          delayed_rt_addr[0] = 0;
          delayed_enable_reg_write[0] = 0;
          temporary_variable = 0;
        end
      endcase
    end
  end
 end
endmodule
```

Simple Fixed 1 Test Bench:

The “Simple fixed 1” testbench module is an environment where we can simulate the inputs of the “Simple fixed 1” module and monitor its outputs to see how successful it is. This test bench includes various tests and cases covering different types of arithmetic and logical operations.

In this testbench module, inputs such as clock signal, reset signal, operation code, instruction format, destination register address, source register values, immediate value, and control flags are defined. These inputs are connected to the same inputs in “Simple fixed 1” module instance (dut). Additionally, the

outputs like data to be written back to the register file, address of the register to be written back, and flag indicating register write operation in the write-back stage are declared.

This test bench initializes the simulation with a reset signal asserted (reset = 1), sets up different test scenarios by modifying the inputs such as the operation code, source register values, and immediate values, and toggles the clock signal to drive the simulation forward. Each of the test cases are executed sequentially, with unique operations being performed and their results observed. The simulation is then paused for 100 time units before stopping the simulation using the “$stop” system task.

Simple Fixed 1 Test Bench Code:

```
/*****************************************************************************************************
 * Module: Simple Fixed 1 Testbench
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------
 * Description:
 *    This test bench module is used to verify the functionality of the Simple_Fixed_1 module. It provides
 *    stimulus to the module inputs and monitors the module outputs to ensure correct behavior. The test bench
 *    includes various test cases covering different arithmetic and logical operations supported by the
 *    Simple_Fixed_1 module.
 *-----------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Operation code
 *   - instr_format: Instruction format
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Source register A
 *   - src_reg_b: Source register B
 *   - store_reg: Store register
 *   - imm_value: Immediate value
 *   - enable_reg_write: Flag indicating register write operation
 *   - branch_is_taken: Flag indicating branch taken
 *-----------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Data to be written back to the register file
```

```
*   - wb_reg_addr: Address of the register to be written back
*   - wb_enable_reg_write: Flag indicating register write operation in WB stage
*-----------------------------------------------------------------------------------------------------
* Internal Signals:
*   - delayed_rt_data: Delayed register data for forwarding
*   - delayed_rt_addr: Delayed register address for forwarding
*   - delayed_enable_reg_write: Delayed register write flag for forwarding
*****************************************************************************************************/

module Simple_Fixed_1_TB ();

  logic clock, reset;

  //Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  logic [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  logic [ 2:0] instr_format;
  // Destination register address
  logic [ 0:6] dest_reg_addr;
  // Values of source registers
  logic [0:127] src_reg_a, src_reg_b, store_reg;
  // Immediate value, truncated based on instruction format
  logic [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  logic enable_reg_write;
  // Flag indicating if a branch was taken

  //Write Back Stage
  // Output value of Stage 3
  logic [0:127] wb_data;
  // Destination register for write back data
  logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  logic wb_enable_reg_write;
```

```
// Indicates whether a branch is taken
logic branch_is_taken;
// Represents the delayed register address
logic delayed_rt_addr;
// Represents the delayed enable register write signal
logic delayed_enable_reg_write;



Simple_Fixed_1 dut (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    store_reg,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

// Set the initial state of the clock to zero
initial clock = 0;

// Toggle the clock value every 5 time units to simulate oscillation
always begin
  #5 clock = ~clock;
end
```

```
initial begin
  reset = 1;
  instr_format = 3'b000;
  // Set the opcode for the Shift Left Halfword (shlh) operation
  op_code = 11'b00001011111;
  // Set the destination register address to $r3
  dest_reg_addr = 7'b0000011;
  // Set the value of source register A, Halfwords: 16'h0010
  src_reg_a = 128'h1A2B3C4D5E6F7A8B9C0D1E2F3A4B5C6;
  // Set the value of source register B, Halfwords: 16'h0001
  src_reg_b = 128'hF0E1D2C3B4A59687766554433221100;
  imm_value = 0;
  enable_reg_write = 1;
  #6;
  // At 11ns, disable the reset, enabling the unit
  reset = 0;
  @(posedge clock);
  #1;
  // Set the opcode for the No Operation (nop) instruction
  op_code = 0;
  @(posedge clock);
  // Add Extended
  #3;
  op_code   = 11'b01101000000;
  src_reg_a = 128'h8C15F2E6A90DC4BF2A7899E8C15F2E6A;
  src_reg_b = 128'h9AF507D38A4E62C711B7D459AF507D38;
  @(posedge clock);
  // Carry Generate
  #3;
  op_code   = 11'b00011000010;
  src_reg_a = 128'h56187F3ED26A950DBF2AC3C56187F3ED;
  src_reg_b = 128'hB6A9C81E57D9AF32F5E483BB6A9C81E5;
  @(posedge clock);
  // Subtract from Extended
  #3;
```

```
op_code   = 11'b01101000001;
src_reg_a = 128'h6D475C3A1809E6ABDC30E126D475C3A1;
src_reg_b = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
@(posedge clock);
// Borrow Generate
#3;
op_code   = 11'b00001000010;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
src_reg_b = 128'h8F6D3BA24C7159FACF3F08C8F6D3BA24;
@(posedge clock);
// Add Halfword
#3;
op_code   = 11'b00011001000;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
src_reg_b = 128'h00000000000000000000000000000001;
@(posedge clock);
// Add Halfword
src_reg_a = 128'h9AF507D38A4E62C711B7D459AF507D38;
src_reg_b = 128'h0A21DECB7EF548D1D36E7860A21DECB7;
@(posedge clock);
#3;
op_code   = 11'b00011001000;
// Add Halfword
src_reg_a = 128'h849F36E1D4B5A2FC94DC931849F36E1D;
src_reg_b = 128'hE58179AFDB023865EEABE3DE58179AFD;
@(posedge clock);
#3;
src_reg_a = 128'h6A81F2CD45B796EDC7B2E996A81F2CD4;
@(posedge clock);
//sfh rt, ra, rb : Subtract from Halfword
#3;
op_code   = 11'b00001001000;
src_reg_a = 128'h478D6AC01257E3FBA7C3DEA478D6AC01;
src_reg_b = 128'h3B7912D6EFD84AB41021A683B7912D6E;
@(posedge clock);
```

```
//sfh rt, ra, rb : Subtract from Halfword
#3;
op_code   = 11'b00001001000;
src_reg_a = 128'hFD45F8F5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h8EBA1945D1C7F26378EBA1945D1C7F26;
@(posedge clock);
//sf rt, ra, rb : Subtract from Word
#3;
op_code   = 11'b00001000000;
src_reg_a = 128'h6D475C3A1809E6ABDC30E126D475C3A1;
src_reg_b = 128'hDFA5C9B3824FA71DB7B2EC5DFA5C9B38;
@(posedge clock);
//sf rt, ra, rb : Subtract from Word
#3;
op_code   = 11'b00001000000;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
src_reg_b = 128'h91BE7CAE5F34D2A4DB1E38C91BE7CAE5;
@(posedge clock);
//sfhi rt, ra, imm10 : Subtract from Halfword Immediate
#3;
op_code = 8'b00001101;
instr_format = 4;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
imm_value = 10'b0011111111;
@(posedge clock);
//sfhi rt, ra, imm10 : Subtract from Halfword Immediate
#3;
op_code = 8'b00001101;
instr_format = 4;
src_reg_a = 128'h0F3A9C15E6D7A8C34C27C930F3A9C15E;
imm_value = 10'b1011111111;
@(posedge clock);
//sfi rt, ra, imm10 : Subtract from Word Immediate
#3;
op_code = 8'b00001100;
```

```
instr_format = 4;
src_reg_a = 128'h4228000040647AE1BFC00000BB83126F;
imm_value = 10'b0011111111;
@(posedge clock);
//sfi rt, ra, imm10 : Subtract from Word Immediate
#3;
op_code = 8'b00001100;
src_reg_a = 128'h6246BA3C1485D99AC471226246BA3C14;
imm_value = 10'b1011111111;
instr_format = 4;
@(posedge clock);
//Add Word
#3;
op_code = 11'b00011000000;
instr_format = 0;
src_reg_a = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h3C2FA791D68BFE92EDD6B7B3C2FA791D;
@(posedge clock);
//Add Word
#3;
op_code   = 11'b00011000000;
src_reg_a = 128'h0A21DECB7EF548D1D36E7860A21DECB7;
src_reg_b = 128'h849F36E1D4B5A2FC94DC931849F36E1D;
@(posedge clock);
//ahi rt, ra, imm10 : Add Halfword Immediate
#3;
op_code = 8'b00011101;
instr_format = 4;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
imm_value = 10'b0011111111;
@(posedge clock);
//ai rt, ra, imm10 : Add Word Immediate
#3;
op_code = 8'b00011100;
instr_format = 4;
```

```
src_reg_a = 128'h8F6D3BA24C7159FACF3F08C8F6D3BA24;
imm_value = 10'b1011111111;
@(posedge clock);
instr_format = 3'b000;
// Handling the "AND" operation (and)
#3;
op_code   = 11'b00011000001;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
src_reg_b = 128'h3F87EDD295C61BAE46EF23B3F87EDD29;
@(posedge clock);
// Handling the "OR" operation (or)
#3;
op_code   = 11'b00001000001;
src_reg_a = 128'h4228000040647AE1BFC00000BB83126F;
src_reg_b = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
@(posedge clock);
// Handling the "XOR" operation (xor)
#3;
op_code = 11'b01001000001;
@(posedge clock);
// Handling the "NAND" operation (nand)
#3;
op_code = 11'b00011001001;
@(posedge clock);
src_reg_a = 128'hE2A719385F2B91CDE89E2A719385F2B9;
src_reg_b = 128'h91BE7CAE5F34D2A4DB1E38C91BE7CAE5;
@(posedge clock);
// ceqh rt, ra, rb Compare Equal Halfword
#3;
op_code   = 11'b01111001000;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
src_reg_b = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
@(posedge clock);
// ceq rt, ra, rb Compare Equal Word
#3;
```

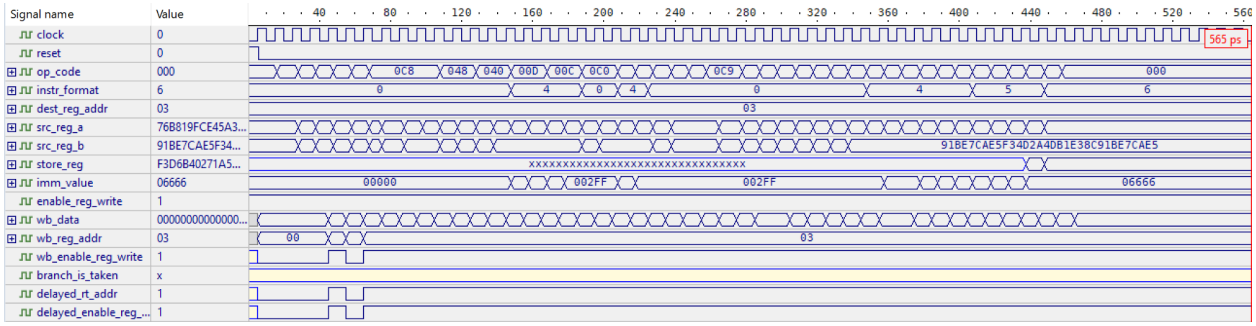
```
op_code   = 11'b01111000000;
src_reg_a = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h8C15F2E6A90DC4BF2A7899E8C15F2E6A;
@(posedge clock);
// cgth rt, ra, rb Compare Greater Than Halfword
#3;
op_code   = 11'b01001001000;
src_reg_a = 128'h6246BA3C1485D99AC471226246BA3C14;
src_reg_b = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
@(posedge clock);
// cgt rt, ra, rb Compare Greater Than Word
#3;
op_code   = 11'b01001000000;
src_reg_a = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h91BE7CAE5F34D2A4DB1E38C91BE7CAE5;
@(posedge clock);
// clgtb rt, ra, rb Compare Logical Greater Than Byte
#3;
op_code   = 11'b01011010000;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
src_reg_b = 128'h849F36E1D4B5A2FC94DC931849F36E1D;
@(posedge clock);
// clgth rt, ra, rb Compare Logical Greater Than Halfword
#3;
op_code   = 11'b01011001000;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
src_reg_b = 128'h3C2FA791D68BFE92EDD6B7B3C2FA791D;
@(posedge clock);
// clgt rt, ra, rb Compare Logical Greater Than Word
#3;
op_code   = 11'b01011000000;
src_reg_a = 128'hBF80DDF5A0CDE28F50E78A5A23F8F5A0;
src_reg_b = 128'h91BE7CAE5F34D2A4DB1E38C91BE7CAE5;
@(posedge clock);
// ceqhi rt, ra, imm10 Compare Equal Halfword Immediate
```

```
#3;
op_code = 8'b01111101;
instr_format = 4;
src_reg_a = 128'h294C7FAEB5D281AEC35F191294C7FAEB;
imm_value = 10'b1011111111;
@(posedge clock);
// ceqi rt, ra, imm10 Compare Equal Word Immediate
#3;
op_code = 8'b01111100;
instr_format = 4;
src_reg_a = 128'h849F36E1D4B5A2FC94DC931849F36E1D;
imm_value = 10'b1111111111;
@(posedge clock);
// cgthi rt, ra, imm10 Compare Greater Than Halfword Immediate
#3;
op_code = 8'b01001101;
instr_format = 4;
src_reg_a = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
imm_value = 10'b1111111111;
@(posedge clock);
// cgti rt, ra, imm10 Compare Greater Than Word Immediate
#3;
op_code = 8'b01001100;
instr_format = 4;
src_reg_a = 128'h3C2FA791D68BFE92EDD6B7B3C2FA791D;
imm_value = 10'b0111111111;
@(posedge clock);
// clgtbi rt, ra, imm10 Compare Logical Greater Than Byte Immediate
#3;
op_code = 8'b01011110;
instr_format = 4;
src_reg_a = 128'h57A8B910243E75FD124ED85957A8B910;
imm_value = 10'b1101111111;
@(posedge clock);
// clgthi rt, ra, imm10 Compare Logical Greater Than Halfword Immediate
```

```
#3;
op_code = 8'b01011101;
instr_format = 4;
src_reg_a = 128'h1D964C0EF27AB9C9A62ED4F91D964C0E;
imm_value = 10'b0111111111;
@(posedge clock);
// ilh rt, imm16 Immediate Load Halfword
#3;
op_code = 9'b010000011;
instr_format = 5;
src_reg_a = 128'hA3F1706B924E3D8B1C9F0B85A3F1706B;
imm_value = 16'b0110011001100110;
@(posedge clock);
// Immediate Load Halfword Upper
#3;
op_code = 9'b010000010;
instr_format = 5;
src_reg_a = 128'hF5E892A65B7C341F0E6F7DA6F5E892A6;
imm_value = 16'b1111111111111110;
@(posedge clock);
// Immediate Load Word
#3;
op_code = 9'b010000001;
instr_format = 5;
src_reg_a = 128'hD8A21F3B5690C7E51D3BF48BD8A21F3B;
imm_value = 32'b11111111111111111111111111111110;
@(posedge clock);
// iohl rt, imm16 Immediate Or Halfword Lower
#3;
op_code = 9'b011000001;
instr_format = 5;
src_reg_a = 128'h2E7F46C1583A9B04AD76A8C72E7F46C1;
imm_value = 16'b0110011001100110;
store_reg = 128'hBC30291F745ED6A58765DDE4BC30291F;
@(posedge clock);
```

```
    // ila rt, imm18 Immediate Load Address
    #3;
    op_code = 7'b0100001;
    instr_format = 6;
    src_reg_a = 128'h76B819FCE45A3D76A5D49F9F76B819FC;
    imm_value = 18'b000110011001100110;
    store_reg = 128'hF3D6B40271A5C8E14FD38C1EF3D6B402;
    @(posedge clock);
    #3;
    op_code = 0;
    @(posedge clock);
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #100;
    op_code = 11'b00000000000;
    $stop;
  end
endmodule
```

Simple Fixed 1 Waveform:



Simple Fixed 2:

Simple Fixed 2:

“Simple fixed 2” module is designed to perform arithmetic and logical operations based on the given instructions and inputs. It consists of separate stages for the register file/ forwarding and write back.

In the register file/ forwarding stage, the module receives inputs like clock signal, reset signal, operation code, instruction format, destination register address, source register values, immediate value, and flags indicating register write operation and branch taken. It will then process these inputs to perform arithmetic operations or logical operations, storing the results in the staging registers for further processing.

In the module also it includes a write back stage where the calculated data is then written back to the register file. This stage helps with using the correct data to write it to the appropriate destination register based on the operation that was performed. Additionally, the module handles different cases where no operation is specified (nop) or when a branch is taken, making sure the proper handling of control flow is in the processor pipeline.

Simple Fixed 2 Code:

```
/*****************************************************************************************************
 * Module: Simple Fixed 2
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------
 * Description:
 *    This module implements a simple fixed-point arithmetic unit. It performs various arithmetic and logical
 *    operations based on the given instructions and inputs. The module contains separate stages for register
 *    file/forwarding and write back.
 *-----------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Operation code
 *   - instr_format: Instruction format
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Source register A
 *   - src_reg_b: Source register B
 *   - imm_value: Immediate value
 *   - enable_reg_write: Flag indicating register write operation
 *   - branch_is_taken: Flag indicating branch taken
 *-----------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Data to be written back to the register file
 *   - wb_reg_addr: Address of the register to be written back
 *   - wb_enable_reg_write: Flag indicating register write operation in WB stage
 *-----------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - delayed_rt_data: Delayed register data for forwarding
 *   - delayed_rt_addr: Delayed register address for forwarding
 *   - delayed_enable_reg_write: Delayed register write flag for forwarding
```

```
**********************************************************************************************************/

module Simple_Fixed_2 (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

  input clock, reset;

  //Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  input [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  input [2:0] instr_format;
  // Destination register address
  input [0:6] dest_reg_addr;
  // Values of source registers
  input [0:127] src_reg_a, src_reg_b;
  // Immediate value, truncated based on instruction format
  input [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  input enable_reg_write;
  // Flag indicating if a branch was taken
  input branch_is_taken;

  //Write Back Stage
  // Output value of Stage 3
  output logic [0:127] wb_data;
  // Destination register for write back data
  output logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic wb_enable_reg_write;

  //Internal Signals
  // Staging register for calculated values
  logic [3:0][0:127] delayed_rt_data;
  // Destination register for write back data
  output logic [3:0][0:6] delayed_rt_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic [3:0] delayed_enable_reg_write;

  // 7-bit counter used for loops
  logic [6:0] i;
  // A temporary variable used for intermediate computations
  logic [0:127] temporary_variable, s;

  always_comb begin
    wb_data = delayed_rt_data[2];
```

```
  wb_reg_addr = delayed_rt_addr[2];
  wb_enable_reg_write = delayed_enable_reg_write[2];
end

always_ff @(posedge clock) begin
  if (reset == 1) begin
    delayed_rt_data[3] <= 0;
    delayed_rt_addr[3] <= 0;
    delayed_enable_reg_write[3] <= 0;
    for (i = 0; i < 3; i = i + 1) begin
      delayed_rt_data[i] <= 0;
      delayed_rt_addr[i] <= 0;
      delayed_enable_reg_write[i] <= 0;
    end
  end else begin
    delayed_rt_data[3] <= delayed_rt_data[2];
    delayed_rt_addr[3] <= delayed_rt_addr[2];
    delayed_enable_reg_write[3] <= delayed_enable_reg_write[2];

    delayed_rt_data[2] <= delayed_rt_data[1];
    delayed_rt_addr[2] <= delayed_rt_addr[1];
    delayed_enable_reg_write[2] <= delayed_enable_reg_write[1];

    delayed_rt_data[1] <= delayed_rt_data[0];
    delayed_rt_addr[1] <= delayed_rt_addr[0];
    delayed_enable_reg_write[1] <= delayed_enable_reg_write[0];
    //nop : No Operation (Execute)
    if (instr_format == 0 && op_code == 0) begin
      delayed_rt_data[0] <= 0;
      delayed_rt_addr[0] <= 0;
      delayed_enable_reg_write[0] <= 0;
    end else begin
      delayed_rt_addr[0] <= dest_reg_addr;
      delayed_enable_reg_write[0] <= enable_reg_write;
      if (branch_is_taken) begin
        delayed_rt_data[0] <= 0;
        delayed_rt_addr[0] <= 0;
        delayed_enable_reg_write[0] <= 0;
      end else if (instr_format == 0) begin
        case (op_code)
          //shlh : Shift Left Halfword
          11'b00001011111: begin
            // Iterate over each halfword in src_reg_b and compute the shifted result
            for (int i = 0; i < 8; i = i + 1) begin
              // Extract the lower 5 bits of src_reg_b
              bit [4:0] shift_amount;
              shift_amount = src_reg_b[(i*16)+:5];
              // Check if shift_amount is less than 16
              if (shift_amount < 16) begin
                // Perform left shift of src_reg_a by shift_amount
                delayed_rt_data[0][(i*16)+:16] = src_reg_a[(i*16)+:16] << shift_amount;
              end else begin
                // If shift_amount is 16 or greater, set the result to 0
                delayed_rt_data[0][(i*16)+:16] = 16'h0000;
              end
            end
          end
          // shlhi rt, ra, value : Shift Left Halfword Immediate
          11'b00001111111: begin
            for (i = 0; i <= 15; i = i + 2) begin
              s = (imm_value & 7'h1F);
```

```
    temporary_variable = src_reg_a[(i*8)+:16];
    for (int b = 0; b < 16; b = b + 1) begin
      if (b + s < 16) delayed_rt_data[0][(i*8)+b] = temporary_variable[b+s];
      else delayed_rt_data[0][(i*8)+b] = 0;
    end
  end
end
//shl rt, ra, rb : Shift Left Word
11'b00001011011: begin
  // Iterate over each word in src_reg_a
  for (int i = 0; i < 16; i = i + 4) begin
    // Extract the lower 6 bits of src_reg_b to determine the shift amount
    bit [5:0] shift_amount;
    shift_amount = src_reg_b[(i*8)+:32] & 32'h0000003F;
    // Perform left shift if the shift amount is within the word size
    delayed_rt_data[0][(i * 8) +: 32] = (shift_amount < 32) ? (src_reg_a[(i * 8) +: 32] << shift_amount) : 32'h00000000;
  end
end
//roth rt, ra, rb : Rotate Halfword
11'b00001011100: begin
  for (i = 0; i <= 15; i = i + 2) begin
    temporary_variable[0:15] = src_reg_a[(i*8)+:16];
    for (int b = 0; b < 16; b = b + 1) begin
      if ((b + (src_reg_b[(i*8)+:16] & 16'h000F)) < 16) begin
        delayed_rt_data[0][(i*8)+b] = temporary_variable[b+(src_reg_b[(i*8) +: 16] & 16'h000F)];
      end else begin
        delayed_rt_data[0][(i*8)+b] = temporary_variable[b+(src_reg_b[(i*8) +: 16] & 16'h000F)-16];
      end
    end
  end
end
//rot rt, ra, rb : Rotate Word
11'b00001011000: begin
  for (i = 0; i <= 15; i = i + 4) begin
    temporary_variable[0:31] = src_reg_a[(i*8)+:32];
    for (int b = 0; b < 32; b = b + 1) begin
      if ((b + (src_reg_b[(i*8)+:32] & 32'h0000001F)) < 32) begin
        delayed_rt_data[0][(i*8)+b] = temporary_variable[b+(src_reg_b[(i*8) +: 32] & 32'h0000001F)];
      end else begin
        delayed_rt_data[0][(i*8)+b] = temporary_variable[b+(src_reg_b[(i*8) +: 32] & 32'h0000001F)-32];
      end
    end
  end
end
//shli rt, ra, imm7 : Shift Left Word Immediate
11'b00001111011: begin
  // Extract the shift amount from imm_value
  bit [6:0] shift_amount;
  shift_amount = imm_value[11:17];
  // Iterate over each word in src_reg_a
  for (int i = 0; i <= 15; i = i + 4) begin
    // Assign the current word to a temporary variable
    bit [31:0] temporary_variable;
    temporary_variable = src_reg_a[(i*8)+:32];
    // Perform left shift based on the shift_amount
    for (int b = 0; b < 32; b = b + 1) begin
      // Check if the shifted index is within bounds
      if (b + shift_amount < 32)
        delayed_rt_data[0][(i*8)+b] = temporary_variable[b+shift_amount];
      else delayed_rt_data[0][(i*8)+b] = 0;
    end
```

```
          end
        end
        // rothi rt, ra, imm7 : Rotate Halfword Immediate
        11'b00001111100: begin
          // Extract the shift amount from imm_value
          bit [6:0] shift_amount;
          shift_amount = imm_value[11:17];
          // Iterate over each halfword in src_reg_a
          for (int i = 0; i <= 15; i = i + 2) begin
            // Assign the current halfword to a temporary variable
            bit [15:0] temporary_variable;
            temporary_variable = src_reg_a[(i*8)+:16];
            // Perform rotation based on the shift_amount
            for (int b = 0; b < 16; b = b + 1) begin
              // Calculate the rotated index
              int rotated_index;
              rotated_index = (b + (shift_amount & 16'h000F)) % 16;
              // Assign the rotated value to delayed_rt_data
              delayed_rt_data[0][(i*8)+b] = temporary_variable[rotated_index];
            end
          end
        end
        //roti rt, ra, imm7 : Rotate Word Immediate
        11'b00001111000: begin
          // Extract the shift amount from imm_value
          bit [6:0] shift_amount;
          shift_amount = imm_value[11:17];
          // Iterate over each word in src_reg_a
          for (int i = 0; i <= 15; i = i + 4) begin
            // Assign the current word to a temporary variable
            bit [31:0] temporary_variable;
            temporary_variable = src_reg_a[(i*8)+:32];
            // Perform rotation based on the shift_amount
            for (int b = 0; b < 32; b = b + 1) begin
              // Calculate the rotated index
              int rotated_index;
              rotated_index = (b + (shift_amount & 32'h0000001F)) % 32;
              // Assign the rotated value to delayed_rt_data
              delayed_rt_data[0][(i*8)+b] = temporary_variable[rotated_index];
            end
          end
        end
      endcase
    end
  end
 end
endmodule
```

Simple Fixed 2 Test Bench:

“Simple fixed 2” testbench module serves to simulate the behavior of the “Simple fixed 2” module and its arithmetic unit. The testbench is used to test different arithmetic operations and logical operations based on the given instructions and its inputs. It uses separate stages for teh register file/ forwarding and write back, helping to ensure comprehensive validation of the functionality of the module.

In this testbench, the input signals like clock signal, reset signal, operation code, instruction format, destination register address, source register values, immediate value, and flags indicating register write operation and branch taken are provided to the “Simple Fixed 2” module for testing purposes. The testbench sets these inputs in different scenarios, representing various arithmetic and logical operations.

For each of these operations, the testbench sets the correct opcode and input values. Simulation of the executed operations within the Simple Fixed 2 module progresses through different time steps, toggling the clock signal and even updating the inputs accordingly. After the operations are completed, the simulation is then paused for 100 time units before stopping. This is now the end of the test scenario, which using the results we are able to ensure that the processor is reliable with a larger system design and the functionality is correct.

Simple Fixed 2 Test Bench Code:

```
/*****************************************************************************************************
 * Module: Simple Fixed 2 Testbench
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------
 * Description:
 *    This testbench module simulates the behavior of the Simple Fixed 2 module, which implements a simple
 *    fixed-point arithmetic unit. It tests various arithmetic and logical operations based on the given
 *    instructions and inputs. The module contains separate stages for the register file/forwarding and
 *    write back.
 *-----------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Operation code
 *   - instr_format: Instruction format
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Source register A
 *   - src_reg_b: Source register B
 *   - imm_value: Immediate value
 *   - enable_reg_write: Flag indicating register write operation
 *   - branch_is_taken: Flag indicating branch taken
 *-----------------------------------------------------------------------------------------------------
```

```
* Outputs:
*   - wb_data: Data to be written back to the register file
*   - wb_reg_addr: Address of the register to be written back
*   - wb_enable_reg_write: Flag indicating register write operation in WB stage
*-----------------------------------------------------------------------------------------------------
* Internal Signals:
*   - delayed_rt_data: Delayed register data for forwarding
*   - delayed_rt_addr: Delayed register address for forwarding
*   - delayed_enable_reg_write: Delayed register write flag for forwarding
*****************************************************************************************************/

module Simple_Fixed_2_TB ();

  logic clock, reset;

  //Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  logic [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  logic [ 2:0] instr_format;
  // Destination register address
  logic [ 0:6] dest_reg_addr;
  // Values of source registers
  logic [0:127] src_reg_a, src_reg_b;
  // Immediate value, truncated based on instruction format
  logic [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  logic enable_reg_write;

  //Write Back Stage
  // Output value of Stage 3
  logic [0:127] wb_data;
  // Destination register for write back data
  logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
```

```
logic wb_enable_reg_write;
// Indicates whether a branch is taken
logic branch_is_taken;
// Represents the delayed register address
logic delayed_rt_addr;
// Represents the delayed enable register write signal
logic delayed_enable_reg_write;



Simple_Fixed_2 dut (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write
);

// Set the initial state of the clock to zero
initial clock = 0;

// Toggle the clock value every 5 time units to simulate oscillation
always begin
  #5 clock = ~clock;
end
```

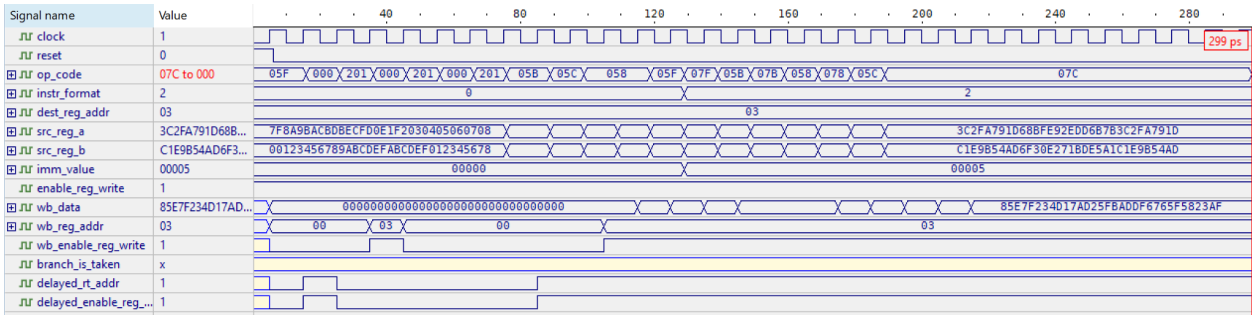
```
initial begin
  reset = 1;
  instr_format = 3'b000;
  // Set the opcode for the Shift Left Halfword (shlh) operation
  op_code = 11'b00001011111;
  // Set the destination register address to $r3
  dest_reg_addr = 7'b0000011;
  // Set the value of source register A, Halfwords: 16'h0010
  src_reg_a = 128'h7F8A9BACBDBECFD0E1F2030405060708;
  // Set the value of source register B, Halfwords: 16'h0001
  src_reg_b = 128'h0123456789ABCDEFABCDEF012345678;
  imm_value = 0;
  enable_reg_write = 1;
  #6;
  // At 11ns, disable the reset, enabling the unit
  reset = 0;
  @(posedge clock);
  #1;
  // Set the opcode for the No Operation (nop) instruction
  op_code = 0;
  @(posedge clock);
  #1;
  op_code = 11'b01000000001;
  @(posedge clock);
  #1;
  op_code = 0;
  @(posedge clock);
  #1;
  op_code = 11'b01000000001;
  @(posedge clock);
  #1;
  op_code = 0;
  @(posedge clock);
  #1;
  op_code = 11'b01000000001;
```

```
@(posedge clock);
#1;
// Handling the "Shift Left Word" operation (shl)
op_code   = 11'b00001011011;
src_reg_b = 128'hFEDCBA9876543210FEDCBA987654321;
src_reg_a = 128'hA1B2C3D4E5F67890A1B2C3D4E5F6789;
@(posedge clock);
#4;
// Handling the "Rotate Halfword" operation (roth)
op_code   = 11'b00001011100;
src_reg_b = 128'h1234567890ABCDEF1234567890ABCDEF;
src_reg_a = 128'hFEDCBA0987654321FEDCBA098765432;
@(posedge clock);
#4;
// Handling the "Rotate Word" operation (rot)
op_code   = 11'b00001011000;
src_reg_b = 128'h9876543210ABCDEF0123456789ABCDEF;
src_reg_a = 128'hFEDCBA9876543210123456789ABCDEF;
@(posedge clock);
#4;
// Handling the "Rotate Word" operation (rot)
op_code   = 11'b00001011000;
src_reg_b = 128'hABCDEFFEDCBA9876543210ABCDEF012;
src_reg_a = 128'h0123456789ABCDEFABCDEF012345678;
@(posedge clock);
#4;
// Shift Left Halfword
op_code   = 11'b00001011111;
src_reg_b = 128'hFEDCBA9876543210FEDCBA9876543210;
src_reg_a = 128'h13579BDF2468ACE02468ACE13579BDF;
@(posedge clock);
#4;
// Shift Left Halfword Immediate
op_code = 11'b00001111111;
instr_format = 3'b010;
```

```
imm_value = 5;
src_reg_b = 128'hFEDCBA9876543210ABCDEF012345678;
src_reg_a = 128'h0123456789ABCDEFABCDEF0123456789;
@(posedge clock);
#4;
// Shift Left Word
op_code   = 11'b00001011011;
src_reg_b = 128'hFEDCBA9876543210FEDCBA987654321;
src_reg_a = 128'hABCDEFFEDCBA9876543210ABCDEFABC;
@(posedge clock);
#4;
// Shift Left Word Immediate
op_code = 11'b00001111011;
instr_format = 3'b010;
imm_value = 5;
src_reg_b = 128'h5A23F8F5A0CDE28F50E78A5A23F8F5A0;
src_reg_a = 128'h3B7912D6EFD84AB41021A683B7912D6E;
@(posedge clock);
#4;
// Rotate Word
op_code   = 11'b00001011000;
src_reg_b = 128'h8EBA1945D1C7F26378EBA1945D1C7F26;
src_reg_a = 128'hF4C65329B71D890F9DC17E8F4C65329B;
@(posedge clock);
#4;
// Rotate Word Immediate
op_code = 11'b00001111000;
instr_format = 3'b010;
imm_value = 5;
src_reg_b = 128'hE2A719385F2B91CDE89E2A719385F2B9;
src_reg_a = 128'h620DF4EAC1723B856BCA55B620DF4EAC;
@(posedge clock);
#4;
// Rotate Halfword
op_code   = 11'b00001011100;
```

```
    src_reg_b = 128'h15781AD7E4B6298F6F90B5F15781AD7E;
    src_reg_a = 128'h478D6AC01257E3FBA7C3DEA478D6AC01;
    @(posedge clock);
    #4;
    // Rotate Halfword Immediate
    op_code = 11'b00001111100;
    instr_format = 3'b010;
    imm_value = 5;
    src_reg_b = 128'hC1E9B54AD6F30E271BDE5A1C1E9B54AD;
    src_reg_a = 128'h3C2FA791D68BFE92EDD6B7B3C2FA791D;
    @(posedge clock);
    #4;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #100;
    op_code = 11'b00000000000;
    $stop;
  end
endmodule
```

Simple Fixed 2 Waveform:



Single Precision:

Single Precision:

“Single Precision” module is single precision arithmetic units designed to perform arithmetic operations based on given instructions and inputs. It uses separate stages for the register file/ forwarding and write back.

In the register file/ forwarding stage, the module takes inputs like clock signals, reset signals, operation codes, instruction formats, register addresses, source register values, immediate values, and flags indicating register write operation and branch taken.This module will then process all these inputs to perform arithmetic operations like addition, subtraction, multiplication, and multiplication with immediate values.

The outputs of this module are data to be written back to the register file, the address of the register to be written back, flags indicating register write operations, and output data from the module. The internal signals in the module handle the staging of the calculated values and flags which will indicate the type of operation being performed.

This module code uses procedural blocks (always_comb and always_ff) to help with the proper execution of operations based on the clock inputs and signals. Arithmetic operations are performed to the specific operation codes and instructions. Using correct handling of the register data and control flags, this module provides an efficient solution for single precision arithmetic operations.

Single Precision Code:

```
/*****************************************************************************************************
 * Module: Single Precision
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------
 * Description:
 *     This module implements a single precision unit. It performs various arithmetic
 *     operations based on the given instructions and inputs. The module contains separate stages for register
 *     file/forwarding and write back.
 *-----------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Operation code
 *   - instr_format: Instruction format
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Source register A
 *   - src_reg_b: Source register B
 *   - temporary_register_c: Temporary register C
 *   - imm_value: Immediate value
 *   - enable_reg_write: Flag indicating register write operation
 *   - branch_is_taken: Flag indicating branch taken
 *-----------------------------------------------------------------------------------------------------
 * Outputs:
 *   - wb_data: Data to be written back to the register file
```

```
 *   - wb_reg_addr: Address of the register to be written back
 *   - wb_enable_reg_write: Flag indicating register write operation in WB stage
 *   - int_data: Output data of the module
 *   - int_reg_addr: Destination register address for int_data
 *   - int_enable_reg_write: Flag indicating register write operation for int_data
 *-----------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - delayed_rt_data: Delayed register data for forwarding
 *   - delayed_rt_addr: Delayed register address for forwarding
 *   - delayed_enable_reg_write: Delayed register write flag for forwarding
 *   - int_operation_flag: Flag indicating the type of operation
 ******************************************************************************************************/

module Single_Precision (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    temporary_register_c,
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    int_data,
    int_reg_addr,
    int_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write,
    int_operation_flag
);

  input clock, reset;

  //Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  input [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  input [2:0] instr_format;
  // Destination register address
  input [0:6] dest_reg_addr;
  // Values of source registers
  input [0:127] src_reg_a, src_reg_b, temporary_register_c;
  // Immediate value, truncated based on instruction format
  input [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
  input enable_reg_write;
  // Flag indicating if a branch was taken
  input branch_is_taken;

  //Write Back Stage
  // Output value of Stage 6
  output logic [0:127] wb_data;
  // Destination register for write back data
  output logic [0:6] wb_reg_addr;
  // Flag indicating if the write back data will be written to the Register Table
  output logic wb_enable_reg_write;
```

```
//Integer Stage
// Output value of Stage 7
output logic [0:127] int_data;
// Destination register for write back data
output logic [0:6] int_reg_addr;
// Flag indicating if the write back data will be written to the Register Table
output logic int_enable_reg_write;

//Internal Signals
// Staging register for calculated values
logic [6:0][0:127] delayed_rt_data;
// Destination register for write back data
output logic [6:0][0:6] delayed_rt_addr;
// Flag indicating if the write back data will be written to the Register Table
output logic [6:0] delayed_enable_reg_write;
// Flag indicating the type of operation
output logic [6:0] int_operation_flag;

always_comb begin
  // Check if the FP7 writeback is enabled for integer operations
  if (int_operation_flag[5] == 1) begin
    int_data = delayed_rt_data[5];
    int_reg_addr = delayed_rt_addr[5];
    int_enable_reg_write = delayed_enable_reg_write[5];
  end else begin
    int_data = 0;
    int_reg_addr = 0;
    int_enable_reg_write = 0;
  end
  // Check if the FP6 writeback is enabled for integer operations
  if (int_operation_flag[4] == 0) begin
    wb_data = delayed_rt_data[4];
    wb_reg_addr = delayed_rt_addr[4];
    wb_enable_reg_write = delayed_enable_reg_write[4];
  end else begin
    wb_data = 0;
    wb_reg_addr = 0;
    wb_enable_reg_write = 0;
  end
end

always_ff @(posedge clock) begin
  integer scale;
  shortreal tempfp;
  logic [0:15] temp16;
  if (reset == 1) begin
    delayed_rt_data[6] <= 0;
    delayed_rt_addr[6] <= 0;
    delayed_enable_reg_write[6] <= 0;
    int_operation_flag[6] <= 0;
    for (int i = 0; i < 6; i = i + 1) begin
      delayed_rt_data[i] <= 0;
      delayed_rt_addr[i] <= 0;
      delayed_enable_reg_write[i] <= 0;
      int_operation_flag[i] <= 0;
    end
  end else begin
    delayed_rt_data[6] <= delayed_rt_data[5];
    delayed_rt_addr[6] <= delayed_rt_addr[5];
    delayed_enable_reg_write[6] <= delayed_enable_reg_write[5];
```

```
int_operation_flag[6] <= int_operation_flag[5];
for (int i = 0; i < 5; i = i + 1) begin
  delayed_rt_data[i+1] <= delayed_rt_data[i];
  delayed_rt_addr[i+1] <= delayed_rt_addr[i];
  delayed_enable_reg_write[i+1] <= delayed_enable_reg_write[i];
  int_operation_flag[i+1] <= int_operation_flag[i];
end
//nop : No Operation (Execute)
if (instr_format == 0 && op_code[0:9] == 0000000000) begin
  delayed_rt_data[0] <= 0;
  delayed_rt_addr[0] <= 0;
  delayed_enable_reg_write[0] <= 0;
  int_operation_flag[0] <= 0;
end else begin
  delayed_rt_addr[0] <= dest_reg_addr;
  delayed_enable_reg_write[0] <= enable_reg_write;
  if (branch_is_taken) begin
    int_operation_flag[0] <= 0;
    delayed_rt_data[0] <= 0;
    delayed_rt_addr[0] <= 0;
    delayed_enable_reg_write[0] <= 0;
  end else if (instr_format == 0) begin
    case (op_code)
      //mpy : Multiply
      11'b01111000100: begin
        int a;
        int b;
        int_operation_flag[0] <= 1;
        for (int i = 0; i < 4; i = i + 1) begin
          a = src_reg_a[(i*32)+31-:16];
          b = src_reg_b[(i*32)+31-:16];
          delayed_rt_data[0][(i*32)+:32] <= $signed(a) * $signed(b);
        end
      end
      //mpyu : Multiply Unsigned
      11'b01111001100: begin
        int unsigned_a;
        int unsigned_b;
        int_operation_flag[0] <= 1;
        for (int i = 0; i < 4; i = i + 1) begin
          unsigned_a = src_reg_a[(i*32)+31-:16];
          unsigned_b = src_reg_b[(i*32)+31-:16];
          delayed_rt_data[0][(i*32)+:32] <= $unsigned(unsigned_a) * $unsigned(unsigned_b);
        end
      end
      //mpyh : Multiply High
      11'b01111000101: begin
        int signed_a;
        int signed_b;
        int_operation_flag[0] <= 1;
        for (int i = 0; i < 4; i = i + 1) begin
          signed_a = $signed(src_reg_a[(i*32)+:16]);
          signed_b = $signed(src_reg_b[(i*32)+16+:16]);
          delayed_rt_data[0][(i*32)+:32] <= (signed_a * signed_b) << 16;
        end
      end
      // mpyhh : Multiply High High
      11'b01111000110: begin
        int signed_a;
        int signed_b;
        int_operation_flag[0] <= 1;
```

```
  for (int i = 0; i < 4; i = i + 1) begin
    signed_a = $signed(src_reg_a[(i*32)+:16]);
    signed_b = $signed(src_reg_b[(i*32)+:16]);
    delayed_rt_data[0][(i*32)+:32] <= (signed_a * signed_b) >> 32;
  end
end
// mpys : Multiply and Shift Right
11'b01111000111: begin
  int signed_a;
  int signed_b;
  int_operation_flag[0] <= 1;
  for (int i = 0; i < 4; i = i + 1) begin
    signed_a = $signed(src_reg_a[(i*32)+31-:16]);
    signed_b = $signed(src_reg_b[(i*32)+31-:16]);
    delayed_rt_data[0][(i*32)+:32] <= ($signed(signed_a * signed_b)) >> 1;
  end
end
//fa : Floating Add
11'b01011000100: begin
  int_operation_flag[0] <= 0;
  for (int i = 0; i < 4; i = i + 1) begin
    real result;  // Declare result variable outside of the loop
    bit [31:0] src_a;
    bit [31:0] src_b;
    // Extract 32-bit long vectors
    src_a  = src_reg_a[(i*32)+:32];
    src_b  = src_reg_b[(i*32)+:32];
    // Perform the calculation
    result = ($bitstoshortreal(src_a) + $bitstoshortreal(src_b));
    // Perform the conditional assignment
    if (result >= $bitstoshortreal(32'h7F7FFFFF))
      delayed_rt_data[0][(i*32)+:32] = 32'h7F7FFFFF;
    else if (result <= $bitstoshortreal(32'hFF7FFFFF))
      delayed_rt_data[0][(i*32)+:32] = 32'hFF7FFFFF;
    else if (result <= $bitstoshortreal(32'h00000001) && result > 0)
      delayed_rt_data[0][(i*32)+:32] = 32'h00000001;
    else if (result >= $bitstoshortreal(32'h80000001) && result < 0)
      delayed_rt_data[0][(i*32)+:32] = 32'h80000001;
    else delayed_rt_data[0][(i*32)+:32] = $shortrealtobits(result);
  end
end
default: begin
  int_operation_flag[0] <= 0;
  delayed_rt_data[0] <= 0;
  delayed_rt_addr[0] <= 0;
  delayed_enable_reg_write[0] <= 0;
end
//fs : Floating Subtract
11'b01011000101: begin
  int_operation_flag[0] <= 0;
  for (int i = 0; i < 4; i = i + 1) begin
    real result;  // Declare result variable outside of the loop
    bit [31:0] src_a;
    bit [31:0] src_b;
    // Extract 32-bit long vectors
    src_a  = src_reg_a[(i*32)+:32];
    src_b  = src_reg_b[(i*32)+:32];
    // Perform the calculation
    result = ($bitstoshortreal(src_a) - $bitstoshortreal(src_b));
    // Perform the conditional assignment
    if (result >= $bitstoshortreal(32'h7F7FFFFF))
```

```
          delayed_rt_data[0][(i*32)+:32] = 32'h7F7FFFFF;
        else if (result <= $bitstoshortreal(32'hFF7FFFFF))
          delayed_rt_data[0][(i*32)+:32] = 32'hFF7FFFFF;
        else if (result <= $bitstoshortreal(32'h00000001) && result > 0)
          delayed_rt_data[0][(i*32)+:32] = 32'h00000001;
        else if (result >= $bitstoshortreal(32'h80000001) && result < 0)
          delayed_rt_data[0][(i*32)+:32] = 32'h80000001;
        else delayed_rt_data[0][(i*32)+:32] = $shortrealtobits(result);
      end
    end
    //fm : Floating Multiply
    11'b01011000110: begin
      int_operation_flag[0] <= 0;
      for (int i = 0; i < 4; i = i + 1) begin
        real result;  // Declare result variable outside of the loop
        bit [31:0] src_a;
        bit [31:0] src_b;
        // Extract 32-bit long vectors
        src_a  = src_reg_a[(i*32)+:32];
        src_b  = src_reg_b[(i*32)+:32];
        // Perform the calculation
        result = ($bitstoshortreal(src_a) * $bitstoshortreal(src_b));
        // Perform the conditional assignment
        if (result >= $bitstoshortreal(32'h7F7FFFFF))
          delayed_rt_data[0][(i*32)+:32] = 32'h7F7FFFFF;
        else if (result <= $bitstoshortreal(32'hFF7FFFFF))
          delayed_rt_data[0][(i*32)+:32] = 32'hFF7FFFFF;
        else if (result <= $bitstoshortreal(32'h00000001) && result > 0)
          delayed_rt_data[0][(i*32)+:32] = 32'h00000001;
        else if (result >= $bitstoshortreal(32'h80000001) && result < 0)
          delayed_rt_data[0][(i*32)+:32] = 32'h80000001;
        else delayed_rt_data[0][(i*32)+:32] = $shortrealtobits(result);
      end
    end
  endcase
end else if (instr_format == 1) begin
  case (op_code[7:10])
    //mpya : Multiply and Add
    4'b1100: begin
      int_operation_flag[0] <= 1;
      for (int i = 0; i < 4; i = i + 1)
        delayed_rt_data[0][(i*32)+:32] <= ($signed(
            src_reg_a[(i*32)+16+:16]
        ) * $signed(
            src_reg_b[(i*32)+16+:16]
        )) + $signed(
            temporary_register_c[(i*32)+:32]
        );
    end
    //fma : Floating Multiply and Add
    4'b1110: begin
      int_operation_flag[0] <= 0;
      for (int i = 0; i < 4; i = i + 1) begin
        real result;  // Declare result variable outside of the loop
        bit [31:0] src_a;
        bit [31:0] src_b;
        bit [31:0] temp_c;
        // Extract 32-bit long vectors
        src_a = src_reg_a[(i*32)+:32];
        src_b = src_reg_b[(i*32)+:32];
        temp_c = temporary_register_c[(i*32)+:32];
```

```
    // Perform the calculation
    result = ($bitstoshortreal(src_a) * $bitstoshortreal(src_b)) +
        $bitstoshortreal(temp_c);
    // Perform the conditional assignment
    if (result >= $bitstoshortreal(32'h7F7FFFFF))
      delayed_rt_data[0][(i*32)+:32] = 32'h7F7FFFFF;
    else if (result <= $bitstoshortreal(32'hFF7FFFFF))
      delayed_rt_data[0][(i*32)+:32] = 32'hFF7FFFFF;
    else if (result <= $bitstoshortreal(32'h00000001) && result > 0)
      delayed_rt_data[0][(i*32)+:32] = 32'h00000001;
    else if (result >= $bitstoshortreal(32'h80000001) && result < 0)
      delayed_rt_data[0][(i*32)+:32] = 32'h80000001;
    else delayed_rt_data[0][(i*32)+:32] = $shortrealtobits(result);
  end
end
//fms : Floating Multiply and Subtract
4'b1111: begin
  int_operation_flag[0] <= 0;
  for (int i = 0; i < 4; i = i + 1) begin
    real result;  // Declare result variable outside of the loop
    bit [31:0] src_a;
    bit [31:0] src_b;
    bit [31:0] temp_c;
    // Extract 32-bit long vectors
    src_a = src_reg_a[(i*32)+:32];
    src_b = src_reg_b[(i*32)+:32];
    temp_c = temporary_register_c[(i*32)+:32];
    // Perform the calculation
    result = ($bitstoshortreal(src_a) * $bitstoshortreal(src_b)) -
        $bitstoshortreal(temp_c);
    // Perform the conditional assignment
    if (result >= $bitstoshortreal(32'h7F7FFFFF))
      delayed_rt_data[0][(i*32)+:32] = 32'h7F7FFFFF;
    else if (result <= $bitstoshortreal(32'hFF7FFFFF))
      delayed_rt_data[0][(i*32)+:32] = 32'hFF7FFFFF;
    else if (result <= $bitstoshortreal(32'h00000001) && result > 0)
      delayed_rt_data[0][(i*32)+:32] = 32'h00000001;
    else if (result >= $bitstoshortreal(32'h80000001) && result < 0)
      delayed_rt_data[0][(i*32)+:32] = 32'h80000001;
    else delayed_rt_data[0][(i*32)+:32] = $shortrealtobits(result);
  end
end
// fnms : Floating Negative Multiply and Subtract
4'b1101: begin
  int_operation_flag[0] <= 0;
  for (int i = 0; i < 4; i = i + 1) begin
    real result;  // Declare result variable outside of the loop
    bit [31:0] src_a;
    bit [31:0] src_b;
    bit [31:0] temp_c;
    // Extract 32-bit long vectors
    src_a = src_reg_a[(i*32)+:32];
    src_b = src_reg_b[(i*32)+:32];
    temp_c = temporary_register_c[(i*32)+:32];
    // Perform the calculation
    result = (-$bitstoshortreal(src_a) * $bitstoshortreal(src_b)) -
        $bitstoshortreal(temp_c);
    // Perform the conditional assignment
    if (result >= $bitstoshortreal(32'h7F7FFFFF))
      delayed_rt_data[0][(i*32)+:32] = 32'h7F7FFFFF;
    else if (result <= $bitstoshortreal(32'hFF7FFFFF))
```

```
            delayed_rt_data[0][(i*32)+:32] = 32'hFF7FFFFF;
           else if (result <= $bitstoshortreal(32'h00000001) && result > 0)
            delayed_rt_data[0][(i*32)+:32] = 32'h00000001;
           else if (result >= $bitstoshortreal(32'h80000001) && result < 0)
            delayed_rt_data[0][(i*32)+:32] = 32'h80000001;
           else delayed_rt_data[0][(i*32)+:32] = $shortrealtobits(result);
          end
         end
         default begin
          int_operation_flag[0] <= 0;
          delayed_rt_data[0] <= 0;
          delayed_rt_addr[0] <= 0;
          delayed_enable_reg_write[0] <= 0;
         end
        endcase
       end else if (instr_format == 3) begin
        case (op_code[1:10])
         default begin
          int_operation_flag[0] <= 0;
          delayed_rt_data[0] <= 0;
          delayed_rt_addr[0] <= 0;
          delayed_enable_reg_write[0] <= 0;
         end
        endcase
       end else if (instr_format == 4) begin
        case (op_code[3:10])
         //mpyi : Multiply Immediate
         8'b01110100: begin
          int_operation_flag[0] <= 1;
          for (int i = 0; i < 4; i = i + 1)
           delayed_rt_data[0][(i*32)+:32] <= $signed(
            src_reg_a[(i*32)+16+:16]
           ) * $signed(
            imm_value[8:17]
           );
         end
         //mpyui : Multiply Unsigned Immediate
         8'b01110101: begin
          int_operation_flag[0] <= 1;
          temp16 = $signed(imm_value[8:17]);
          for (int i = 0; i < 4; i = i + 1)
           delayed_rt_data[0][(i*32)+:32] <= $unsigned(
            src_reg_a[(i*32)+16+:16]
           ) * $unsigned(
            temp16
           );
         end
         default begin
          int_operation_flag[0] <= 0;
          delayed_rt_data[0] <= 0;
          delayed_rt_addr[0] <= 0;
          delayed_enable_reg_write[0] <= 0;
         end
        endcase
       end
      end
     end
    end
   end
endmodule
```

Single Precision Test Bench:

The “Single Precision” testbench serves as verification for the design under the test (dut). This testbench is for inputs in the dut and monitoring its outputs. It will generate various test scenarios to verify correct functionality of the arithmetic units, ensuring that it will perform as expected.

In this testbench, the inputs are clock signals, reset signals, operation codes, instruction formats, register addresses, source register values, immediate values, and flags indicating register write operation and branch taken are defined. These inputs are passed to the dut module which is the start in the testbench.

This testbench initializes the clock signal and toggles its value every 5 time units to simulate oscillation. Also this testbench resets the signal to bring the dut to a known state before starting the test cases. Different unique test cases are then executed sequentially by changing the operation code and other inputs. This allows the testbench to simulate arithmetic operations such as addition, subtraction, multiplication, and more. Finally, the simulation is paused for 200 time units before stopping the simulation to indicate that this is the end of the testbench.

Single Precision Test Bench Code:

```
/*****************************************************************************************************
 * Module: Single Precision Testbench
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------
 * Description:
 *     This module serves as the testbench for the single precision unit. It drives
 *     inputs to the DUT (Design Under Test) and monitors its outputs. The testbench generates various test
 *     scenarios to verify the functionality of the arithmetic unit.
 *-----------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - op_code: Operation code
 *   - instr_format: Instruction format
 *   - dest_reg_addr: Destination register address
 *   - src_reg_a: Source register A
 *   - src_reg_b: Source register B
 *   - temporary_register_c: Temporary register C
```

```
*   - imm_value: Immediate value
*   - enable_reg_write: Flag indicating register write operation
*   - branch_is_taken: Flag indicating branch taken
*-----------------------------------------------------------------------------------------------------
* Outputs:
*   - wb_data: Data to be written back to the register file
*   - wb_reg_addr: Address of the register to be written back
*   - wb_enable_reg_write: Flag indicating register write operation in WB stage
*   - int_data: Output data of the module
*   - int_reg_addr: Destination register address for int_data
*   - int_enable_reg_write: Flag indicating register write operation for int_data
*-----------------------------------------------------------------------------------------------------
* Internal Signals:
*   - delayed_rt_data: Delayed register data for forwarding
*   - delayed_rt_addr: Delayed register address for forwarding
*   - delayed_enable_reg_write: Delayed register write flag for forwarding
*   - int_operation_flag: Flag indicating the type of operation
*****************************************************************************************************/

module Single_Precision_TB ();

  logic clock, reset;

  //Register File/Forwarding Stage
  // Decoded opcode, truncated based on instruction format
  logic [0:10] op_code;
  // Format of instruction, used with opcode and immediate value
  logic [ 2:0] instr_format;
  // Destination register address
  logic [ 0:6] dest_reg_addr;
  // Values of source registers
  logic [0:127] src_reg_a, src_reg_b, temporary_register_c;
  // Immediate value, truncated based on instruction format
  logic [0:17] imm_value;
  // Flag indicating if the current instruction will write to the Register Table
```

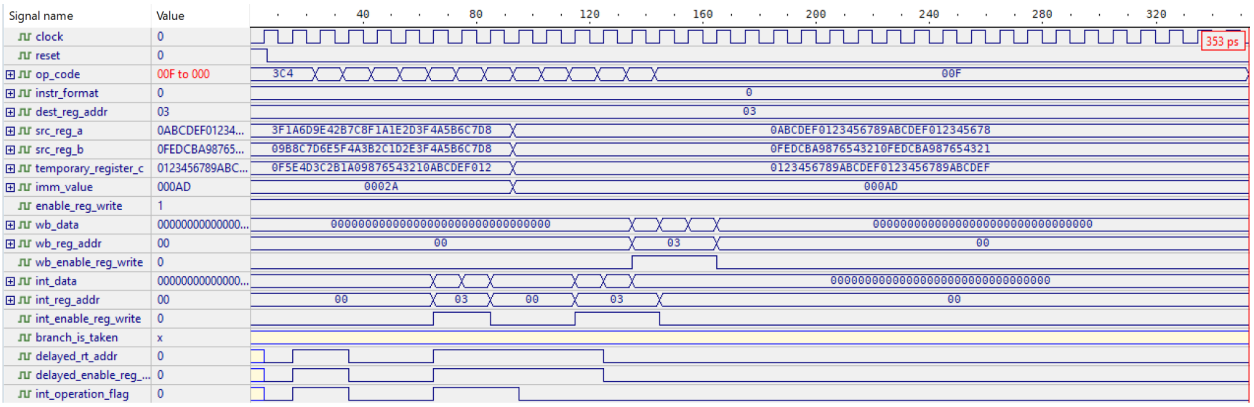
```
logic enable_reg_write;

//Write Back Stage
// Output value of Stage 6
logic [0:127] wb_data;
// Destination register for write back data
logic [0:6] wb_reg_addr;
// Flag indicating if the write back data will be written to the Register Table
logic wb_enable_reg_write;

// Output value of Stage 7
logic [0:127] int_data;
// Destination register for write back data
logic [0:6] int_reg_addr;
// Flag indicating if the write back data will be written to the Register Table
logic int_enable_reg_write;
// Indicates whether a branch is taken
logic branch_is_taken;
// Represents the delayed register address
logic delayed_rt_addr;
// Represents the delayed enable register write signal
logic delayed_enable_reg_write;
// Indicates whether an integer operation is flagged
logic int_operation_flag;


Single_Precision dut (
    clock,
    reset,
    op_code,
    instr_format,
    dest_reg_addr,
    src_reg_a,
    src_reg_b,
    temporary_register_c,
```

```
    imm_value,
    enable_reg_write,
    wb_data,
    wb_reg_addr,
    wb_enable_reg_write,
    int_data,
    int_reg_addr,
    int_enable_reg_write,
    branch_is_taken,
    delayed_rt_addr,
    delayed_enable_reg_write,
    int_operation_flag
);

// Set the initial state of the clock to zero
initial clock = 0;

// Toggle the clock value every 5 time units to simulate oscillation
always begin
  #5 clock = ~clock;
end

initial begin
  reset = 1;
  instr_format = 3'b000;
  // Multiply
  op_code = 11'b01111000100;
  dest_reg_addr = 7'b0000011;
  src_reg_a = 128'h3F1A6D9E42B7C8F1A1E2D3F4A5B6C7D8;
  src_reg_b = 128'h9B8C7D6E5F4A3B2C1D2E3F4A5B6C7D8;
  temporary_register_c = 128'hF5E4D3C2B1A09876543210ABCDEF012;
  imm_value = 42;
  enable_reg_write = 1;
  #6;
  reset = 0;
```

```
@(posedge clock);
#8;
// Multiply Unsigned
op_code = 11'b01111001100;
@(posedge clock);
#8;
// Multiply Immediate
op_code = 8'b01110100;
@(posedge clock);
#8;
// Multiply Unsigned Immediate
op_code = 8'b01110101;
@(posedge clock);
#8;
// Multiply and Add
op_code = 4'b1100;
@(posedge clock);
#8;
// Multiply High
op_code = 11'b01111000101;
@(posedge clock);
#8;
// Multiply and Shift Right
op_code = 11'b01111000111;
@(posedge clock);
#8;
// Multiply High High
op_code = 11'b01111000110;
@(posedge clock);
#8;
// Floating Add
op_code = 11'b01011000100;
dest_reg_addr = 7'b0000011;
src_reg_a = 128'hABCDEF0123456789ABCDEF012345678;
src_reg_b = 128'hFEDCBA9876543210FEDCBA987654321;
```

```
    temporary_register_c = 128'h0123456789ABCDEF0123456789ABCDEF;
    imm_value = 173;
    @(posedge clock);
    #8;
    // Floating Subtract
    op_code = 11'b01011000101;
    @(posedge clock);
    #8;
    // Floating Multiply
    op_code = 11'b01011000110;
    @(posedge clock);
    #8;
    // Floating Multiply and Add
    op_code = 4'b1110;
    @(posedge clock);
    #8;
    // Floating Negative Multiply and Subtract
    op_code = 4'b1101;
    @(posedge clock);
    #8;
    // Floating Multiply and Subtract
    op_code = 4'b1111;
    @(posedge clock);
    #8;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #200;
    op_code = 11'b00000000000;
    $stop;
  end
endmodule
```

Single Precision Waveform:



Decode:

Decode:

The “Decode” modules primary function is to interpret machine instructions, extracting different parameters like opcode, register addresses, immediate values, and execution unit destinations.
This module takes several inputs, including clock signals, reset signals, an array of 32-bit instructions, the current program counter, and signals related to pipeline stall conditions and branch prediction. It also has internal signals that manage the decoding process and track teh state of the pipeline.

The main output of the Decode module is to include signals indicating the pipeline is stalled or not, the next instruction to be executed, and whether a branch is taken. It also provides signals related to the program counter in case of stalls.

This module decodes instructions by understanding the opcode and other characteristics. It understands the difference between even and odd signals in a pair and processes them accordingly. The decoding process involves analyzing different factors like register dependencies to avoid different hazards like RAW errors. This module utilizes a combinatorial logic block for the decoding process, where it can inspect instructions and determine their properties. It also includes sequential logic to handle pipeline stages and manage program counter updates.

Decode Code:

```
/*****************************************************************************************************************
 * Module: Decode
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------------------
 * Description:
 *     This module decodes instructions in a processor, extracting opcode, register addresses, immediate values,
 *     and other relevant information.
 *-----------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - instruction: Array of 32-bit instructions
 *   - program_counter: Current program counter
 *   - stall_program_counter: Stalled program counter
 *   - stall: Signal indicating whether the pipeline is stalled
 *   - branch_is_taken_reg: Signal indicating whether a branch is taken
 *-----------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - stall_program_counter: Stalled program counter
 *   - stall: Signal indicating whether the pipeline is stalled
 *   - branch_is_taken_reg: Signal indicating whether a branch is taken
 *   - instruction_next: Next instruction to be executed in the pipeline
 *-----------------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - instruction_even, instruction_odd: Instructions from the decoder
 *   - op_code_even, op_code_odd: Opcode of instructions
 *   - register_write_even, register_write_odd: Signal indicating whether instructions will write to registers
 *   - immediate_even, immediate_odd: Immediate values
 *   - rt_address_even, rt_address_odd: Destination register addresses
 *   - unit_is_even, unit_is_odd: Destination execution units
 *   - format_is_even, format_is_odd: Instruction formats
 *   - initial_odd_out: Signal indicating whether the odd instruction is the first instruction in a pair
 *   - stall_var: Internal signal indicating pipeline stall
 *   - stall_program_counter_var: Internal signal indicating stalled program counter
 *****************************************************************************************************************/

module Decode (
    clock,
    reset,
    instruction,
    program_counter,
    stall_program_counter,
    stall,
    branch_is_taken_reg
);

  input logic clock, reset;

  input logic [0:31] instruction[0:1];
  logic [0:31] instruction_next[0:1], instruction_dec[0:1], instruction_next_reg[0:1];
  // Instructions received from the decoder
  logic [0:31] instruction_even, instruction_odd, instruction_odd_issue, instruction_even_issue;

  //Signals for handling branches
  // New program counter value for branch operations
  logic [7:0] program_counter_wb;
  output logic [7:0] stall_program_counter;
  output logic stall;
```

```
// Indicates if a branch was taken in the instruction
output logic branch_is_taken_reg;
// Flag to indicate if a branch was taken in the instruction
logic branch_is_taken;
// Indicates if the odd instruction is the first opcode in the pair
logic initial_odd, initial_odd_out;
logic stall_var;
logic [7:0] stall_program_counter_var;

// 8-bit program counter for tracking the current state
input logic [7:0] program_counter;

//Nets from decode logic
// Format of the instruction for even and odd cases
logic [2:0] format_is_even, format_is_odd;
// Opcode of the instruction for even and odd cases (used with instruction_format)
logic [0:10] op_code_even, op_code_odd;
// Destination execution unit of instruction
logic [1:0] unit_is_even, unit_is_odd;
// Destination register addresses for even and odd instructions
logic [0:6] rt_address_even, rt_address_odd;
// Full possible immediate value for even and odd instructions (used with instruction_format)
logic [0:17] immediate_even, immediate_odd;
// Indicates if the even or odd instruction will write to the target register
logic register_write_even, register_write_odd;
// Due to how the 'finished' signal is detected, a workaround is needed to prevent flagging after reset
logic first_cycle;
// New program counter for handling stalls
logic [7:0] program_counter_dec, program_counter_pipe;

localparam [0:31] NOP = 32'b01000000001000000000000000000000;
localparam [0:31] LNOP = 32'b00000000001000000000000000000000;

//Internal Signals for Handling RAW Errors
// Destination register for writeback stage for even and odd instructions
logic [0:6] delay_rt_address_even, delay_rt_address_odd;
// Flags to delay register write for even and odd instructions
logic delay_register_write_even, delay_register_write_odd;
// Flags to indicate if the first or second signal should be stalled due to a Read-After-Write (RAW) Errors
logic raw_first_stall, raw_second_stall;
// Flags to indicate if the first or second signal should be stalled due to a Read-After-Write (RAW) or structural Errors
logic first_stall, second_stall;
// Destination register for writeback stage in the floating point unit 1
logic [6:0][0:6] delay_rt_address_fp1;
// Flags to delay register write in the floating point unit 1
logic [6:0] delay_register_write_fp1;
// Flags to indicate if the floating point unit 1 will write an integer result
logic [6:0] delay_integer_fp1;
// Destination register for writeback stage in the fixed point unit 2
logic [3:0][0:6] delay_rt_address_fx2;
// Flags to delay register write in the fixed point unit 2
logic [3:0] delay_register_write_fx2;
// Destination register for writeback stage in the byte unit 1
logic [3:0][0:6] delay_rt_address_b1;
// Flags to delay register write in the byte unit 1
logic [3:0] delay_register_write_b1;
// Destination register for writeback stage in the fixed point unit 1
logic [1:0][0:6] delay_rt_address_fx1;
// Flags to delay register write in the fixed point unit 1
logic [1:0] delay_register_write_fx1;
// Destination register for writeback stage in the permute unit 1
```

```
logic [3:0][0:6] delay_rt_address_p1;
// Flags to delay register write in the permute unit 1
logic [3:0] delay_register_write_p1;
// Destination register for writeback stage in the load/store unit 1
logic [5:0][0:6] delay_rt_address_ls1;
// Flags to delay register write in the load/store unit 1
logic [5:0] delay_register_write_ls1;

typedef struct {
  logic [0:31] instruction;
  logic [0:10] op_code;
  logic reg_write;
  // Flags to indicate if the even and odd instructions are valid
  logic is_even_valid, is_odd_valid;
  logic [0:17] immediate;
  logic [0:6] target_address;
  logic [1:0] execution_unit;
  logic [2:0] instruction_format;
  logic [0:6] source_a_address, source_b_address, source_c_address;
  // Flags to indicate if the source registers A, B, and C are read in this instruction
  logic source_a_valid, source_b_valid, source_c_valid;
} op_code;

op_code first_opcode, second_opcode;

Pipes pipe (
    .clock(clock),
    .reset(reset),
    .program_counter(program_counter_pipe),
    .instruction_even(instruction_even),
    .instruction_odd(instruction_odd),
    .program_counter_wb(program_counter_wb),
    .branch_is_taken(branch_is_taken),
    .op_code_even(op_code_even),
    .op_code_odd(op_code_odd),
    .unit_is_even(unit_is_even),
    .unit_is_odd(unit_is_odd),
    .rt_address_even(rt_address_even),
    .rt_address_odd(rt_address_odd),
    .format_is_even(format_is_even),
    .format_is_odd(format_is_odd),
    .immediate_even(immediate_even),
    .immediate_odd(immediate_odd),
    .register_write_even(register_write_even),
    .register_write_odd(register_write_odd),
    .initial_odd(initial_odd_out),
    .delay_rt_address_even(delay_rt_address_even),
    .delay_register_write_even(delay_register_write_even),
    .delay_rt_address_odd(delay_rt_address_odd),
    .delay_register_write_odd(delay_register_write_odd),
    .delay_rt_address_fp1(delay_rt_address_fp1),
    .delay_register_write_fp1(delay_register_write_fp1),
    .delay_integer_fp1(delay_integer_fp1),
    .delay_rt_address_fx2(delay_rt_address_fx2),
    .delay_register_write_fx2(delay_register_write_fx2),
    .delay_rt_address_b1(delay_rt_address_b1),
    .delay_register_write_b1(delay_register_write_b1),
    .delay_rt_address_fx1(delay_rt_address_fx1),
    .delay_register_write_fx1(delay_register_write_fx1),
    .delay_rt_address_p1(delay_rt_address_p1),
    .delay_register_write_p1(delay_register_write_p1),
```

```
    .delay_rt_address_ls1(delay_rt_address_ls1),
    .delay_register_write_ls1(delay_register_write_ls1)
);

always_ff @(posedge clock) begin : decode_op
  if (first_opcode.is_even_valid) begin
    instruction_even <= first_opcode.instruction;
    op_code_even <= first_opcode.op_code;
    register_write_even <= first_opcode.reg_write;
    immediate_even <= first_opcode.immediate;
    rt_address_even <= first_opcode.target_address;
    unit_is_even <= first_opcode.execution_unit;
    format_is_even <= first_opcode.instruction_format;
    initial_odd_out <= 0;
  end else if (second_opcode.is_even_valid) begin
    instruction_even <= second_opcode.instruction;
    op_code_even <= second_opcode.op_code;
    register_write_even <= second_opcode.reg_write;
    immediate_even <= second_opcode.immediate;
    rt_address_even <= second_opcode.target_address;
    unit_is_even <= second_opcode.execution_unit;
    format_is_even <= second_opcode.instruction_format;
  end else begin
    instruction_even <= 0;
    op_code_even <= 0;
    register_write_even <= 0;
    immediate_even <= 0;
    rt_address_even <= 0;
    unit_is_even <= 0;
    format_is_even <= 0;
  end
  if (first_opcode.is_odd_valid) begin
    instruction_odd <= first_opcode.instruction;
    op_code_odd <= first_opcode.op_code;
    register_write_odd <= first_opcode.reg_write;
    immediate_odd <= first_opcode.immediate;
    rt_address_odd <= first_opcode.target_address;
    unit_is_odd <= first_opcode.execution_unit;
    format_is_odd <= first_opcode.instruction_format;
    initial_odd_out <= 1;
  end else if (second_opcode.is_odd_valid) begin
    instruction_odd <= second_opcode.instruction;
    op_code_odd <= second_opcode.op_code;
    register_write_odd <= second_opcode.reg_write;
    immediate_odd <= second_opcode.immediate;
    rt_address_odd <= second_opcode.target_address;
    unit_is_odd <= second_opcode.execution_unit;
    format_is_odd <= second_opcode.instruction_format;
    initial_odd_out <= 0;
  end else begin
    instruction_odd <= 0;
    op_code_odd <= 0;
    register_write_odd <= 0;
    immediate_odd <= 0;
    rt_address_odd <= 0;
    unit_is_odd <= 0;
    format_is_odd <= 0;
  end
  instruction_next_reg <= instruction_next;
  stall <= stall_var;
  stall_program_counter <= stall_program_counter_var;
```

```
  branch_is_taken_reg <= branch_is_taken;
  program_counter_pipe <= program_counter_dec;
  // Flag is always high after reset and low otherwise
  first_cycle <= reset;
 end

 //Decode logic
 always_comb begin
  first_stall  = 0;
  second_stall = 0;
  if (reset == 1) begin
   stall_var = 0;
   stall_program_counter_var = 0;
   initial_odd = 0;
   first_opcode = inspect_one(0);
   second_opcode = inspect_one(0);
   instruction_next[0] = 0;
   instruction_next[1] = 0;
  end else begin
   if (branch_is_taken == 1) begin
    stall_program_counter_var = program_counter_wb + 2;
    stall_var = 1;
    instruction_next[0] = 0;
    instruction_next[1] = 0;
    first_opcode = inspect_one(0);
    second_opcode = inspect_one(0);
   end else begin
    if (stall == 1) begin
     instruction_dec[0] = instruction_next_reg[0];
     instruction_dec[1] = instruction_next_reg[1];
     stall_var = 0;
     program_counter_dec = stall_program_counter;
    end else begin
     instruction_dec[0]  = instruction[0];
     instruction_dec[1]  = instruction[1];
     program_counter_dec = program_counter;
    end
    if ((instruction_dec[0] != 0)) begin
     // Checking if the first opcode instruction is present in the decoded instruction
     first_opcode = inspect_one(instruction_dec[0]);
     if (first_opcode.source_a_valid) begin
      for (int i = 0; i <= 4; i++) begin
       if ((first_opcode.source_a_address == delay_rt_address_fp1[i]) && (delay_register_write_fp1[i] && delay_integer_fp1[i])) begin
        first_stall = 1;
       end
       if ((i < 4) &&
                                                                    (((first_opcode.source_a_address ==
delay_rt_address_ls1[i]) && delay_register_write_ls1[i]) ||
                                                                    ((first_opcode.source_a_address ==
delay_rt_address_fp1[i]) && delay_register_write_fp1[i]))) begin
        first_stall = 1;
       end
       if ((i < 2) &&
                                                                    (((first_opcode.source_a_address ==
delay_rt_address_fx2[i]) && delay_register_write_fx2[i]) ||
                                                                    ((first_opcode.source_a_address ==
delay_rt_address_b1[i]) && delay_register_write_b1[i]) ||
                                                                    ((first_opcode.source_a_address ==
delay_rt_address_p1[i]) && delay_register_write_p1[i]))) begin
        first_stall = 1;
       end
```

```
        end
        if ((first_opcode.source_a_address == delay_rt_address_even) && delay_register_write_even) begin
          first_stall = 1;
        end
        if ((first_opcode.source_a_address == delay_rt_address_odd) && delay_register_write_odd) begin
          first_stall = 1;
        end
      end
      if (first_opcode.source_b_valid) begin
        for (int i = 0; i <= 4; i++) begin
          if ((first_opcode.source_b_address == delay_rt_address_fp1[i]) && (delay_register_write_fp1[i] && delay_integer_fp1[i])) begin
            first_stall = 1;
          end
          if ((i < 4) &&
                                                                    (((first_opcode.source_b_address ==
delay_rt_address_ls1[i]) && delay_register_write_ls1[i]) ||
                                                                    ((first_opcode.source_b_address ==
delay_rt_address_fp1[i]) && delay_register_write_fp1[i]))) begin
            first_stall = 1;
          end
          if ((i < 2) &&
                                                                    (((first_opcode.source_b_address ==
delay_rt_address_fx2[i]) && delay_register_write_fx2[i]) ||
                                                                    ((first_opcode.source_b_address ==
delay_rt_address_b1[i]) && delay_register_write_b1[i]) ||
                                                                    ((first_opcode.source_b_address ==
delay_rt_address_p1[i]) && delay_register_write_p1[i]))) begin
            first_stall = 1;
          end
        end
        if ((first_opcode.source_b_address == delay_rt_address_even) && delay_register_write_even) begin
          first_stall = 1;
        end
        if ((first_opcode.source_b_address == delay_rt_address_odd) && delay_register_write_odd) begin
          first_stall = 1;
        end
      end
      if (first_opcode.source_c_valid) begin
        for (int i = 0; i <= 4; i++) begin
          if ((first_opcode.source_c_address == delay_rt_address_fp1[i]) && (delay_register_write_fp1[i] && delay_integer_fp1[i])) begin
            first_stall = 1;
          end

          if ((i < 4) &&
                                                                    (((first_opcode.source_c_address ==
delay_rt_address_ls1[i]) && delay_register_write_ls1[i]) ||
                                                                    ((first_opcode.source_c_address ==
delay_rt_address_fp1[i]) && delay_register_write_fp1[i]))) begin
            first_stall = 1;
          end
          if ((i < 2) &&
                                                                    (((first_opcode.source_c_address ==
delay_rt_address_fx2[i]) && delay_register_write_fx2[i]) ||
                                                                    ((first_opcode.source_c_address ==
delay_rt_address_b1[i]) && delay_register_write_b1[i]) ||
                                                                    ((first_opcode.source_c_address ==
delay_rt_address_p1[i]) && delay_register_write_p1[i]))) begin
            first_stall = 1;
          end
        end
        if ((first_opcode.source_c_address == delay_rt_address_even) && delay_register_write_even) begin
```

```
          first_stall = 1;
        end
        if ((first_opcode.source_c_address == delay_rt_address_odd) && delay_register_write_odd) begin
          first_stall = 1;
        end
      end
    end else begin
      first_opcode = inspect_one(0);
    end
    if ((instruction_dec[1] != 0)) begin
      second_opcode = inspect_one(instruction_dec[1]);
      // Checking for structural Errors. If both the first and second opcode are valid in the same pipeline (even or odd), a structural Error occurs.
      if ((second_opcode.is_even_valid && first_opcode.is_even_valid) || (second_opcode.is_odd_valid && first_opcode.is_odd_valid)) begin
        second_stall = 1;
      end
                                                    // Checking for Write-After-Write (WAW) data Error. If both the first and second opcode
are writing to the same register, a WAW Error occurs.
      else if (first_opcode.reg_write && second_opcode.reg_write && (first_opcode.target_address == second_opcode.target_address)) begin
        second_stall = 1;
      end else begin
        if (second_opcode.source_a_valid) begin
          for (int i = 0; i <= 4; i++) begin
            if ((second_opcode.source_a_address == delay_rt_address_fp1[i]) && (delay_register_write_fp1[i] && delay_integer_fp1[i])) begin
              second_stall = 1;
            end
            if ((i < 4) &&
                                                                    (((second_opcode.source_a_address ==
delay_rt_address_ls1[i]) && delay_register_write_ls1[i]) ||
                                                                    ((second_opcode.source_a_address ==
delay_rt_address_fp1[i]) && delay_register_write_fp1[i]))) begin
              second_stall = 1;
            end
            if ((i < 2) &&
                                                                    (((second_opcode.source_a_address ==
delay_rt_address_fx2[i]) && delay_register_write_fx2[i]) ||
                                                                    ((second_opcode.source_a_address ==
delay_rt_address_b1[i]) && delay_register_write_b1[i]) ||
                                                                    ((second_opcode.source_a_address ==
delay_rt_address_p1[i]) && delay_register_write_p1[i]))) begin
              second_stall = 1;
            end
          end
          if ((second_opcode.source_a_address == delay_rt_address_even) && delay_register_write_even) begin
            second_stall = 1;
          end
          if ((second_opcode.source_a_address == delay_rt_address_odd) && delay_register_write_odd) begin
            second_stall = 1;
          end
          if ((second_opcode.source_a_address == first_opcode.target_address) && first_opcode.reg_write) begin
            second_stall = 1;
          end
        end
        if (second_opcode.source_b_valid) begin
          for (int i = 0; i <= 4; i++) begin
            if ((second_opcode.source_b_address == delay_rt_address_fp1[i]) && (delay_register_write_fp1[i] && delay_integer_fp1[i])) begin
              second_stall = 1;
            end
            if ((i < 4) &&
                                                                    (((second_opcode.source_b_address ==
delay_rt_address_ls1[i]) && delay_register_write_ls1[i]) ||
```

```
                                                                                ((second_opcode.source_b_address ==
delay_rt_address_fp1[i]) && delay_register_write_fp1[i]))) begin
                second_stall = 1;
              end
              if ((i < 2) &&
                                                                                (((second_opcode.source_b_address ==
delay_rt_address_fx2[i]) && delay_register_write_fx2[i]) ||
                                                                                ((second_opcode.source_b_address ==
delay_rt_address_b1[i]) && delay_register_write_b1[i]) ||
                                                                                ((second_opcode.source_b_address ==
delay_rt_address_p1[i]) && delay_register_write_p1[i]))) begin
                second_stall = 1;
              end
            end
            if ((second_opcode.source_b_address == delay_rt_address_even) && delay_register_write_even) begin
              second_stall = 1;
            end
            if ((second_opcode.source_b_address == delay_rt_address_odd) && delay_register_write_odd) begin
              second_stall = 1;
            end
            if ((second_opcode.source_b_address == first_opcode.target_address) && first_opcode.reg_write) begin
              second_stall = 1;
            end
          end
          if (second_opcode.source_c_valid) begin
            for (int i = 0; i <= 4; i++) begin
              if ((second_opcode.source_c_address == delay_rt_address_fp1[i]) && (delay_register_write_fp1[i] && delay_integer_fp1[i])) begin
                second_stall = 1;
              end
              if ((i < 4) &&
                                                                                (((second_opcode.source_c_address ==
delay_rt_address_ls1[i]) && delay_register_write_ls1[i]) ||
                                                                                ((second_opcode.source_c_address ==
delay_rt_address_fp1[i]) && delay_register_write_fp1[i]))) begin
                second_stall = 1;
              end
              if ((i < 2) &&
                                                                                (((second_opcode.source_c_address ==
delay_rt_address_fx2[i]) && delay_register_write_fx2[i]) ||
                                                                                ((second_opcode.source_c_address ==
delay_rt_address_b1[i]) && delay_register_write_b1[i]) ||
                                                                                ((second_opcode.source_c_address ==
delay_rt_address_p1[i]) && delay_register_write_p1[i]))) begin
                second_stall = 1;
              end
            end
            if ((second_opcode.source_c_address == delay_rt_address_even) && delay_register_write_even) begin
              second_stall = 1;
            end
            if ((second_opcode.source_c_address == delay_rt_address_odd) && delay_register_write_odd) begin
              second_stall = 1;
            end
            if ((second_opcode.source_c_address == first_opcode.target_address) && first_opcode.reg_write) begin
              second_stall = 1;
            end
          end
        end
      end else begin
        second_opcode = inspect_one(0);
      end
      if (first_stall) begin
```

```
        stall_program_counter_var = program_counter_dec;
        stall_var = 1;
        instruction_next[0] = instruction_dec[0];
        instruction_next[1] = instruction_dec[1];
        first_opcode = inspect_one(0);
        second_opcode = inspect_one(0);
      end else if (second_stall) begin
        stall_program_counter_var = program_counter_dec;
        stall_var = 1;
        instruction_next[0] = 0;
        instruction_next[1] = instruction_dec[1];
        second_opcode = inspect_one(0);
      end else begin
        stall_var = 0;
      end
    end
  end
end

function op_code inspect_one(input logic [0:31] instruction);
  if (reset == 1) begin
    inspect_one.op_code = 0;
    inspect_one.is_even_valid = 1;
    inspect_one.is_odd_valid = 1;
  end
  //Even decoding
  inspect_one.reg_write = 1;
  inspect_one.target_address = instruction[25:31];
  inspect_one.source_a_address = instruction[18:24];
  inspect_one.source_b_address = instruction[11:17];
  inspect_one.source_c_address = instruction[25:31];
  inspect_one.instruction = instruction;
  inspect_one.is_even_valid = 1;
  //alternate nop
  if (instruction == 0) begin
    inspect_one.instruction_format = 0;
    inspect_one.source_a_valid = 0;
    inspect_one.source_b_valid = 0;
    inspect_one.source_c_valid = 0;
    inspect_one.op_code = 0;
    inspect_one.execution_unit = 0;
    inspect_one.target_address = 0;
    inspect_one.immediate = 0;
    inspect_one.reg_write = 0;
    inspect_one.is_even_valid = 0;
    // End of RRR-type instructions handling
  end  // Handling the "Floating Multiply and Add" operation (fma)
  else if (instruction[0:3] == 4'b1110) begin
    inspect_one.instruction_format = 1;
    inspect_one.source_a_valid = 1;
    inspect_one.source_b_valid = 1;
    inspect_one.source_c_valid = 1;
    inspect_one.op_code = 4'b1110;
    inspect_one.execution_unit = 0;
    inspect_one.target_address = instruction[4:10];
  end  // Handling the "Floating Negative Multiply and Subtract" operation (fnms)
  else if (instruction[0:3] == 4'b1101) begin
    inspect_one.instruction_format = 1;
    inspect_one.source_a_valid = 1;
    inspect_one.source_b_valid = 1;
    inspect_one.source_c_valid = 1;
```

```
  inspect_one.op_code = 4'b1101;
  inspect_one.execution_unit = 0;
  inspect_one.target_address = instruction[4:10];
end  // Handling the "Floating Multiply and Subtract" operation (fms)
else if (instruction[0:3] == 4'b1111) begin
  inspect_one.instruction_format = 1;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 1;
  inspect_one.source_c_valid = 1;
  inspect_one.op_code = 4'b1111;
  inspect_one.execution_unit = 0;
  inspect_one.target_address = instruction[4:10];
  // End of RI18-type instructions handling
end  // Handling the "Immediate Load Address" operation (ila)
else if (instruction[0:6] == 7'b0100001) begin
  inspect_one.instruction_format = 6;
  inspect_one.source_a_valid = 0;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 7'b0100001;
  inspect_one.execution_unit = 3;
  inspect_one.immediate = $signed(instruction[7:24]);
  // End of RI8-type instructions handling
end  //                  // Handling the "Add Extended" operation (addx)
else if (instruction[0:10] == 11'b01101000000) begin
  inspect_one.instruction_format = 0;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 1;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 11'b01101000000;
  inspect_one.execution_unit = 0;
  inspect_one.immediate = 0;
end  //                  // Handling the "Carry Generate" operation (cg)
else if (instruction[0:11] == 11'b00011000010) begin
  inspect_one.instruction_format = 0;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 1;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 11'b00011000010;
  inspect_one.execution_unit = 0;
  inspect_one.immediate = 0;
end  //                  // Handling the "Borrow Generate" operation (bg)
else if (instruction[0:11] == 11'b00001000010) begin
  inspect_one.instruction_format = 0;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 1;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 11'b00001000010;
  inspect_one.execution_unit = 0;
  inspect_one.immediate = 0;
end  //                  // Handling the "Subtract from Extended" operation (sfx)
else if (instruction[0:11] == 11'b01101000001) begin
  inspect_one.instruction_format = 3;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 11'b01101000001;
  inspect_one.execution_unit = 0;
  inspect_one.immediate = $signed(instruction[10:17]);
  // End of RI16-type instructions handling
end  // Handling the "Immediate Load Halfword" operation (ilh)
```

```
else if (instruction[0:8] == 9'b010000011) begin
  inspect_one.instruction_format = 5;
  inspect_one.source_a_valid = 0;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 9'b010000011;
  inspect_one.execution_unit = 3;
  inspect_one.immediate = $signed(instruction[9:24]);
end  // Handling the "Immediate Load Word" operation (il)
else if (instruction[0:8] == 9'b010000001) begin
  inspect_one.instruction_format = 5;
  inspect_one.source_a_valid = 0;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 9'b010000001;
  inspect_one.execution_unit = 3;
  inspect_one.immediate = $signed(instruction[9:24]);
end  // Handling the "Immediate Load Halfword Upper" operation (ilhu)
else if (instruction[0:8] == 9'b010000010) begin
  inspect_one.instruction_format = 5;
  inspect_one.source_a_valid = 0;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 9'b010000010;
  inspect_one.execution_unit = 3;
  inspect_one.immediate = $signed(instruction[9:24]);
end  // Handling the "Immediate OR Halfword Lower" operation (iohl)
else if (instruction[0:8] == 9'b011000001) begin
  inspect_one.instruction_format = 5;
  inspect_one.source_a_valid = 0;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 9'b011000001;
  inspect_one.execution_unit = 3;
  inspect_one.immediate = $signed(instruction[9:24]);
end else begin
  // Setting the instruction format for RI10-type instructions
  inspect_one.instruction_format = 4;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.immediate = $signed(instruction[8:17]);
  case (instruction[0:7])
    // Handling the "Multiply Immediate" operation (mpyi)
    8'b01110100: begin
      inspect_one.op_code = 8'b01110100;
      inspect_one.execution_unit = 0;
    end
    // Handling the "Multiply Unsigned Immediate" operation (mpyui)
    8'b01110101: begin
      inspect_one.op_code = 8'b01110101;
      inspect_one.execution_unit = 0;
    end
    // Handling the "Add Halfword Immediate" operation (ahi)
    8'b00011101: begin
      inspect_one.op_code = 8'b00011101;
      inspect_one.execution_unit = 3;
    end
    // Handling the "Add Word Immediate" operation (ai)
    8'b00011100: begin
      inspect_one.op_code = 8'b00011100;
```

```
    inspect_one.execution_unit = 3;
  end
  // Handling the "Subtract from Halfword Immediate" operation (sfhi)
  8'b00001101: begin
    inspect_one.op_code = 8'b00001101;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Subtract from Word Immediate" operation (sfi)
  8'b00001100: begin
    inspect_one.op_code = 8'b00001100;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Equal Halfword Immediate" operation (ceqhi)
  8'b01111101: begin
    inspect_one.op_code = 8'b01111101;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Equal Word Immediate" operation (ceqi)
  8'b01111100: begin
    inspect_one.op_code = 8'b01111100;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Greater Than Halfword Immediate" operation (cgthi)
  8'b01001101: begin
    inspect_one.op_code = 8'b01001101;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Greater Than Word Immediate" operation (cgti)
  8'b01001100: begin
    inspect_one.op_code = 8'b01001100;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Logical Greater Than Byte Immediate" operation (clgtbi)
  8'b01011110: begin
    inspect_one.op_code = 8'b01011110;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Logical Greater Than Halfword Immediate" operation (clgthi)
  8'b01011101: begin
    inspect_one.op_code = 8'b01011101;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Logical Greater Than Word Immediate" operation (clgti)
  8'b01011100: begin
    inspect_one.op_code = 8'b01011100;
    inspect_one.execution_unit = 3;
  end
  default: inspect_one.instruction_format = 7;
endcase
if (inspect_one.instruction_format == 7) begin
  // Setting the instruction format for RR-type instructions
  inspect_one.instruction_format = 0;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 1;
  inspect_one.source_c_valid = 0;
  case (instruction[0:10])
    // Handling the "Multiply" operation (mpy)
    11'b01111000100: begin
      inspect_one.op_code = 11'b01111000100;
      inspect_one.execution_unit = 0;
    end
```

```
// Handling the "Multiply and Add" operation (mpya)
4'b1100: begin
  inspect_one.op_code = 4'b1100;
  inspect_one.execution_unit = 0;
end
// Handling the "Multiply and Shift Right" operation (mpys)
11'b01111000111: begin
  inspect_one.op_code = 11'b01111000111;
  inspect_one.execution_unit = 0;
end
// Handling the "Multiply High High" operation (mpyhh)
11'b01111000110: begin
  inspect_one.op_code = 11'b01111000110;
  inspect_one.execution_unit = 0;
end
// Handling the "Multiply Unsigned" operation (mpyu)
11'b01111001100: begin
  inspect_one.op_code = 11'b01111001100;
  inspect_one.execution_unit = 0;
end
// Handling the "Multiply High" operation (mpyh)
11'b01111000101: begin
  inspect_one.op_code = 11'b01111000101;
  inspect_one.execution_unit = 0;
end
// Handling the "Floating Add" operation (fa)
11'b01011000100: begin
  inspect_one.op_code = 11'b01011000100;
  inspect_one.execution_unit = 0;
end
// Handling the "Floating Subtract" operation (fs)
11'b01011000101: begin
  inspect_one.op_code = 11'b01011000101;
  inspect_one.execution_unit = 0;
end
// Handling the "Floating Multiply" operation (fm)
11'b01011000110: begin
  inspect_one.op_code = 11'b01011000110;
  inspect_one.execution_unit = 0;
end
// Handling the "Shift Left Halfword" operation (shlh)
11'b00001011111: begin
  inspect_one.op_code = 11'b00001011111;
  inspect_one.execution_unit = 1;
end
// Handling the "Shift Left Word" operation (shl)
11'b00001011011: begin
  inspect_one.op_code = 11'b00001011011;
  inspect_one.execution_unit = 1;
end
// Handling the "Rotate Halfword" operation (roth)
11'b00001011100: begin
  inspect_one.op_code = 11'b00001011100;
  inspect_one.execution_unit = 1;
end
// Handling the "Rotate Word" operation (rot)
11'b00001011000: begin
  inspect_one.op_code = 11'b00001011000;
  inspect_one.execution_unit = 1;
end
// Handling the "Count Ones in Bytes" operation (cntb)
```

```
11'b01010110100: begin
  inspect_one.op_code = 11'b01010110100;
  inspect_one.execution_unit = 2;
  inspect_one.source_b_valid = 0;
end
// Handling the "Average Bytes" operation (avgb)
11'b00011010011: begin
  inspect_one.op_code = 11'b00011010011;
  inspect_one.execution_unit = 2;
end
// Handling the "Absolute Differences of Bytes" operation (absdb)
11'b00001010011: begin
  inspect_one.op_code = 11'b00001010011;
  inspect_one.execution_unit = 2;
end
// Handling the "Sum Bytes into Halfwords" operation (sumb)
11'b01001010011: begin
  inspect_one.op_code = 11'b01001010011;
  inspect_one.execution_unit = 2;
end
// Handling the "Add Halfword" operation (ah)
11'b00011001000: begin
  inspect_one.op_code = 11'b00011001000;
  inspect_one.execution_unit = 3;
end
// Handling the "Add Word" operation (a)
11'b00011000000: begin
  inspect_one.op_code = 11'b00011000000;
  inspect_one.execution_unit = 3;
end
// Handling the "Subtract from Halfword" operation (sfh)
11'b00001001000: begin
  inspect_one.op_code = 11'b00001001000;
  inspect_one.execution_unit = 3;
end
// Handling the "Subtract from Word" operation (sf)
11'b00001000000: begin
  inspect_one.op_code = 11'b00001000000;
  inspect_one.execution_unit = 3;
end
// Handling the "AND" operation (and)
11'b00011000001: begin
  inspect_one.op_code = 11'b00011000001;
  inspect_one.execution_unit = 3;
end
// Handling the "OR" operation (or)
11'b00001000001: begin
  inspect_one.op_code = 11'b00001000001;
  inspect_one.execution_unit = 3;
end
// Handling the "XOR" operation (xor)
11'b01001000001: begin
  inspect_one.op_code = 11'b01001000001;
  inspect_one.execution_unit = 3;
end
// Handling the "NAND" operation (nand)
11'b00011001001: begin
  inspect_one.op_code = 11'b00011001001;
  inspect_one.execution_unit = 3;
end
// Handling the "Compare Equal Halfword" operation (ceqh)
```

```
  11'b01111001000: begin
    inspect_one.op_code = 11'b01111001000;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Equal Word" operation (ceq)
  11'b01111000000: begin
    inspect_one.op_code = 11'b01111000000;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Greater Than Halfword" operation (cgth)
  11'b01001001000: begin
    inspect_one.op_code = 11'b01001001000;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Greater Than Word" operation (cgt)
  11'b01001000000: begin
    inspect_one.op_code = 11'b01001000000;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Logical Greater Than Byte" operation (clgtb)
  11'b01011010000: begin
    inspect_one.op_code = 11'b01011010000;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Logical Greater Than Halfword" operation (clgth)
  11'b01011001000: begin
    inspect_one.op_code = 11'b01011001000;
    inspect_one.execution_unit = 3;
  end
  // Handling the "Compare Logical Greater Than Word" operation (clgt)
  11'b01011000000: begin
    inspect_one.op_code = 11'b01011000000;
    inspect_one.execution_unit = 3;
  end
  // Handling the "No Operation" (Execute) operation (nop)
  11'b01000000001: begin
    inspect_one.op_code = 11'b01000000001;
    inspect_one.execution_unit = 0;
    inspect_one.reg_write = 0;
  end
  // Handling the "No Operation" (Load) operation (lnop)
  11'b00000000001: begin
    inspect_one.op_code = 11'b00000000001;
    inspect_one.execution_unit = 0;
    inspect_one.reg_write = 0;
  end
  default: inspect_one.instruction_format = 7;
endcase
if (inspect_one.instruction_format == 7) begin
  // Setting the instruction format for RI7-type instructions
  inspect_one.instruction_format = 2;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.immediate = $signed(instruction[11:17]);
  case (instruction[0:10])
    // Handling the "Shift Left Word Immediate" operation (shli)
    11'b00001111011: begin
      inspect_one.op_code = 11'b00001111011;
      inspect_one.execution_unit = 1;
    end
```

```
      // Handling the "Shift Left Halfword Immediate" operation (shlhi)
      11'b00001111111: begin
        inspect_one.op_code = 11'b00001111111;
        inspect_one.execution_unit = 1;
      end
      // Handling the "Rotate Halfword Immediate" operation (rothi)
      11'b00001111100: begin
        inspect_one.op_code = 11'b00001111100;
        inspect_one.execution_unit = 1;
      end
      // Handling the "Rotate Word Immediate" operation (roti)
      11'b00001111000: begin
        inspect_one.op_code = 11'b00001111000;
        inspect_one.execution_unit = 1;
      end
      default begin
        inspect_one.instruction_format = 0;
        inspect_one.source_a_valid = 0;
        inspect_one.source_b_valid = 0;
        inspect_one.source_c_valid = 0;
        inspect_one.op_code = 0;
        inspect_one.execution_unit = 0;
        inspect_one.target_address = 0;
        inspect_one.immediate = 0;
        inspect_one.is_even_valid = 0;
      end
    endcase
  end
 end
end
// Handling odd decoding
if (inspect_one.is_even_valid == 0) begin
 inspect_one.target_address = instruction[25:31];
 inspect_one.source_a_address = instruction[18:24];
 inspect_one.source_b_address = instruction[11:17];
 inspect_one.source_c_address = instruction[25:31];
 inspect_one.reg_write = 1;
 inspect_one.is_odd_valid = 1;
 if (instruction == 0) begin
  inspect_one.instruction_format = 0;
  inspect_one.source_a_valid = 0;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 0;
  inspect_one.execution_unit = 0;
  inspect_one.target_address = 0;
  inspect_one.immediate = 0;
  inspect_one.reg_write = 0;
  inspect_one.is_odd_valid = 0;
  // Handling the end of RI10-type instructions
 end  // Handling the "Load Quadword (d-form)" operation (lqd)
 else if (instruction[0:7] == 8'b00110100) begin
  inspect_one.instruction_format = 4;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.op_code = 8'b00110100;
  inspect_one.execution_unit = 1;
  inspect_one.immediate = $signed(instruction[8:17]);
 end  // Handling the "Store Quadword (d-form)" operation (stqd)
 else if (instruction[0:7] == 8'b00100100) begin
```

```
  inspect_one.instruction_format = 4;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 1;
  inspect_one.op_code = 8'b00100100;
  inspect_one.execution_unit = 1;
  inspect_one.immediate = $signed(instruction[8:17]);
  inspect_one.reg_write = 0;
end else begin
  // Setting the instruction format for RI16-type instructions
  inspect_one.instruction_format = 5;
  inspect_one.source_a_valid = 0;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.immediate = $signed(instruction[9:24]);
  case (instruction[0:8])
    // Handling the "Load Quadword (a-form)" operation (lqa)
    9'b001100001: begin
      inspect_one.op_code = 9'b001100001;
      inspect_one.execution_unit = 1;
    end
    // Handling the "Store Quadword (a-form)" operation (stqa)
    9'b001000001: begin
      inspect_one.op_code = 9'b001000001;
      inspect_one.execution_unit = 1;
      inspect_one.reg_write = 0;
      inspect_one.source_c_valid = 1;
    end
    // Handling the "Load Quadword Instruction Relative (a-form)" operation (lqr)
    9'b001100111: begin
      inspect_one.op_code = 9'b001100111;
      inspect_one.execution_unit = 1;
      inspect_one.reg_write = 0;
      inspect_one.source_c_valid = 1;
    end
    // Handling the "Store Quadword Instruction Relative (a-form)" operation (stqr)
    9'b001000111: begin
      inspect_one.op_code = 9'b001000111;
      inspect_one.execution_unit = 1;
      inspect_one.reg_write = 0;
      inspect_one.source_c_valid = 1;
    end
    // Handling the "Branch Relative" operation (br)
    9'b001100100: begin
      inspect_one.op_code = 9'b001100100;
      inspect_one.execution_unit = 2;
      inspect_one.reg_write = 0;
    end
    // Handling the "Branch Absolute" operation (bra)
    9'b001100000: begin
      inspect_one.op_code = 9'b001100000;
      inspect_one.execution_unit = 2;
      inspect_one.reg_write = 0;
    end
    // Handling the "Branch Indirect If Not Zero" operation (binz)
    11'b00100101001: begin
      inspect_one.op_code = 11'b00100101001;
      inspect_one.execution_unit = 2;
      inspect_one.reg_write = 0;
    end
    // Handling the "Branch Indirect If Not Zero Halfword" operation (binzh)
```

```
  11'b00100101011: begin
    inspect_one.op_code = 11'b00100101011;
    inspect_one.execution_unit = 2;
    inspect_one.reg_write = 0;
  end
  // Handling the "Branch If Zero Halfword" operation (brhz)
  9'b001000100: begin
    inspect_one.op_code = 9'b001000100;
    inspect_one.execution_unit = 2;
    inspect_one.reg_write = 0;
  end
  // Handling the "Branch Absolute and Set Link" operation (brasl)
  9'b001100010: begin
    inspect_one.op_code = 9'b001100010;
    inspect_one.execution_unit = 2;
    inspect_one.reg_write = 0;
  end
  // Handling the "Branch Relative and Set Link" operation (brsl)
  9'b001100110: begin
    inspect_one.op_code = 9'b001100110;
    inspect_one.execution_unit = 2;
  end
  // Handling the "Branch If Not Zero Word" operation (brnz)
  9'b001000010: begin
    inspect_one.op_code = 9'b001000010;
    inspect_one.execution_unit = 2;
    inspect_one.reg_write = 0;
    inspect_one.source_c_valid = 1;
  end
  // Handling the "Branch If Not Zero Halfword" operation (brnzh)
  9'b001000110: begin
    inspect_one.op_code = 9'b001000110;
    inspect_one.execution_unit = 2;
    inspect_one.reg_write = 0;
    inspect_one.source_c_valid = 1;
  end
  // Handling the "Branch If Zero Word" operation (brz)
  9'b001000000: begin
    inspect_one.op_code = 9'b001000000;
    inspect_one.execution_unit = 2;
    inspect_one.reg_write = 0;
    inspect_one.source_c_valid = 1;
  end
  default: inspect_one.instruction_format = 7;
endcase
if (inspect_one.instruction_format == 7) begin
  // Setting the instruction format for RR-type instructions
  inspect_one.instruction_format = 0;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 1;
  inspect_one.source_c_valid = 0;
  case (instruction[0:10])
    // Handling the "Shift Left Quadword by Bits" operation (shlqbi)
    11'b00111011011: begin
      inspect_one.op_code = 11'b00111011011;
      inspect_one.execution_unit = 0;
    end
    // Handling the "Shift Left Quadword by Bytes" operation (shlqby)
    11'b00111011111: begin
      inspect_one.op_code = 11'b00111011111;
      inspect_one.execution_unit = 0;
```

```
  end
  // Handling the "Rotate Quadword by Bytes" operation (rotqby)
  11'b00111011100: begin
    inspect_one.op_code = 11'b00111011100;
    inspect_one.execution_unit = 0;
  end
  // Handling the "Load Quadword (x-form)" operation (lqx)
  11'b00111000100: begin
    inspect_one.op_code = 11'b00111000100;
    inspect_one.execution_unit = 1;
  end
  // Handling the "Store Quadword (x-form)" operation (stqx)
  11'b00101000100: begin
    inspect_one.op_code = 11'b00101000100;
    inspect_one.execution_unit = 1;
    inspect_one.reg_write = 0;
    inspect_one.source_c_valid = 1;
  end
  // Handling the "Branch Indirect" operation (bi)
  11'b00110101000: begin
    inspect_one.op_code = 11'b00110101000;
    inspect_one.execution_unit = 2;
    inspect_one.reg_write = 0;
    inspect_one.source_b_valid = 0;
  end
  // Handling the "Branch Indirect If Zero" operation (biz)
  11'b00100101000: begin
    inspect_one.op_code = 11'b00100101000;
    inspect_one.execution_unit = 2;
    inspect_one.reg_write = 0;
    inspect_one.source_b_valid = 0;
  end
  // Handling the "Branch Indirect If Zero Halfword" operation (bihz)
  11'b00100101010: begin
    inspect_one.op_code = 11'b00100101010;
    inspect_one.execution_unit = 2;
    inspect_one.reg_write = 0;
    inspect_one.source_b_valid = 0;
  end
  default: inspect_one.instruction_format = 7;
endcase
if (inspect_one.instruction_format == 7) begin
  // Setting the instruction format for RI7-type instructions
  inspect_one.instruction_format = 2;
  inspect_one.source_a_valid = 1;
  inspect_one.source_b_valid = 0;
  inspect_one.source_c_valid = 0;
  inspect_one.immediate = $signed(instruction[11:17]);
  case (instruction[0:10])
    // Handling the "Shift Left Quadword by Bits" operation (shlqbi)
    11'b00111011011: begin
      inspect_one.op_code = 11'b00111011011;
      inspect_one.execution_unit = 0;
    end
    // Handling the "Shift Left Quadword by Bits Immediate" operation (shlqbii)
    11'b00111111011: begin
      inspect_one.op_code = 11'b00111111011;
      inspect_one.execution_unit = 0;
    end
    // Handling the "Shift Left Quadword by Bytes from Bit Shift Count" operation (shlqbybi)
    11'b00111001111: begin
```

```
            inspect_one.op_code = 11'b00111001111;
            inspect_one.execution_unit = 0;
          end
          // Handling the "Rotate Quadword by Bytes from Bit Shift Count" operation (rotqbybi)
          11'b00111001100: begin
            inspect_one.op_code = 11'b00111001100;
            inspect_one.execution_unit = 0;
          end
          // Handling the "Shift Left Quadword by Bytes Immediate" operation (shlqbyi)
          11'b00111111111: begin
            inspect_one.op_code = 11'b00111111111;
            inspect_one.execution_unit = 0;
          end
          // Handling the "Rotate Quadword by Bytes Immediate" operation (rotqbyi)
          11'b00111111100: begin
            inspect_one.op_code = 11'b00111111100;
            inspect_one.execution_unit = 0;
          end
          default begin
            inspect_one.instruction_format = 0;
            inspect_one.source_a_valid = 0;
            inspect_one.source_b_valid = 0;
            inspect_one.source_c_valid = 0;
            inspect_one.op_code = 0;
            inspect_one.execution_unit = 0;
            inspect_one.target_address = 0;
            inspect_one.immediate = 0;
            inspect_one.is_odd_valid = 0;
          end
        endcase
      end
    end
  end
 end else inspect_one.is_odd_valid = 0;
endfunction
endmodule
```

Instruction Fetch:

Instruction Fetch:

The “Instruction Fetch” module handles the retrieval of instructions form the instruction cache and manages the program counter. Its main function is to ensure a continuous flow of instructions to the decoding stage while also handling stall condition and branch predictions

Inputs to this module include clock signals, reset signals, the instruction cache containing 64-byte instructions, and signals indicating whether a branch was taken. The primary outputs are signals indicating the readiness to read the next set of instructions and the current program counter value.

This module maintains an instruction line buffer to store fetched instructions. It also manages signals to control the flow of instruction fetching, like stall flags to halt fetching new instructions when necessary.

The module operates in conjunction with the “Decode” stage, where it sends two instructions at a time fro decoding. It also adjusts the program counter accordingly, ensuring that instructions are fetched in sequence and handling cases of branch prediction.

During each clock cycle, this module updates the program counter and retrieves instructions from the cache based on the current program counter values. It also will handle all stall conditions by either continuing the fetch instructions or pausing it until the stall condition is finished.

Instruction Fetch Code:

```
/*****************************************************************************************************************
 * Module: Instruction Fetch
 * Author: Noah Merone
 *-----------------------------------------------------------------------------------------------------------------
 * Description:
 *     This module fetches instructions from the instruction cache and sends them to the Decode stage. It also
 *     manages the program counter (program_counter) and controls the flow of instruction fetching.
 *-----------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - instruction_cache: Instruction cache containing 64B instructions
 *   - branch_is_taken: Signal indicating if a branch was taken
 *-----------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - read_enable: Signal indicating if the Instruction Fetch unit is ready to read the next set of instructions
 *   - program_counter: Current program counter value
 *-----------------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - instr_decode: Array storing the fetched instructions
 *   - stall: Signal indicating if the Instruction Fetch unit should stall fetching new instructions
 *   - program_counter_wb: Program counter to be used as the reset program_counter signal for Instruction Fetch to start reading new instructions
 *   - program_counter_check: Integer used for checkpointing to adjust the program counter while reading from instruction_cache
 *****************************************************************************************************************/

module Instruction_Fetch (
   clock,
   reset,
   instruction_cache,
   program_counter,
   read_enable
);

  input logic clock;
  input logic reset;

  // Instruction line buffer, stores 64B instruction
  input logic [0:31] instruction_cache[0:255];
  // 2 instructions sent to DECODE stage
  logic [0:31] instr_decode[0:1];
  // Flag to indicate a stall. When set, Instruction fetch should stop fetching new instruction
  logic stall;
  // Flag to indicate if a branch was taken
  logic branch_is_taken;
  // Flag to signal that IF is ready to read next set of 64B instruction
  output logic read_enable;
```

```
// Program Counter
output logic [7:0] program_counter;
// program_counter_wb acts as reset program_counter signal for IF to start new instruction read position
logic [7:0] program_counter_wb;
// Used for checkpointing, to adjust the program_counter while reading from instruction_cache
integer program_counter_check;

localparam [0:31] NOP = 32'b01000000001000000000000000000000;
localparam [0:31] LNOP = 32'b00000000001000000000000000000000;

Decode decode (
    clock,
    reset,
    instr_decode,
    program_counter,
    program_counter_wb,
    stall,
    branch_is_taken
);

always_comb begin : program_counter_counter
  if (reset == 1) begin
    program_counter_check = 0;
    read_enable = 1;
  end else begin
    read_enable = 0;
  end
end

always_ff @(posedge clock) begin : fetch_instruction
  if (reset == 1) begin
    program_counter <= 0;
    instr_decode[0] <= 32'h0000;
    instr_decode[1] <= 32'h0000;
  end else begin
    // The 'stall' flag is used to halt the Instruction Fetch (IF) from fetching new instructions.
    // This is particularly useful in cases of dual issue conflicts where the decode stage
    // inserts a No Operation (nop) instruction.
    // The 'program_counter_wb' is then used to resume fetching a new stream of instructions.
    if (stall == 0) begin
      // stall<=0;
      instr_decode[0] <= instruction_cache[program_counter];
      instr_decode[1] <= instruction_cache[program_counter+1];
      if (program_counter < 254) program_counter <= program_counter + 2;
    end else begin
      if (branch_is_taken == 0) begin
        instr_decode[0] <= instruction_cache[program_counter_wb];
        instr_decode[1] <= instruction_cache[program_counter_wb+1];
      end else begin
        program_counter <= program_counter_wb & ~1;
        // If branching to an odd number instruction, indicated by the least significant bit of program_counter_wb being 1
        if (program_counter_wb & 256'h1) begin
          instr_decode[0] <= instruction_cache[program_counter_wb-2];
          instr_decode[1] <= 0;
        end else begin
          instr_decode[0] <= instruction_cache[program_counter_wb-2];
          instr_decode[1] <= instruction_cache[program_counter_wb-1];
        end
      end
    end
  end
```

```
  end
endmodule
```

Instruction Fetch Test Bench:

This module is made to help understand the instruction fetch module and ensure its functionality by providing simulation and monitoring the outputs.

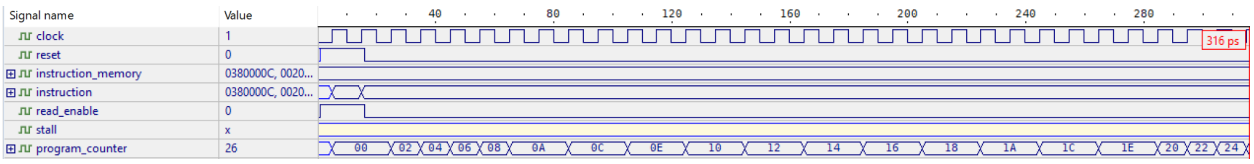
Inputs to this testbench include clock signals, reset signals, and an instruction cache containing 64-byte instructions. Additionally, it takes a signal indicating whether a branch was taken. Outputs from the testbench include signals indicating the readiness to read the next set of instructions and the current program counter value.

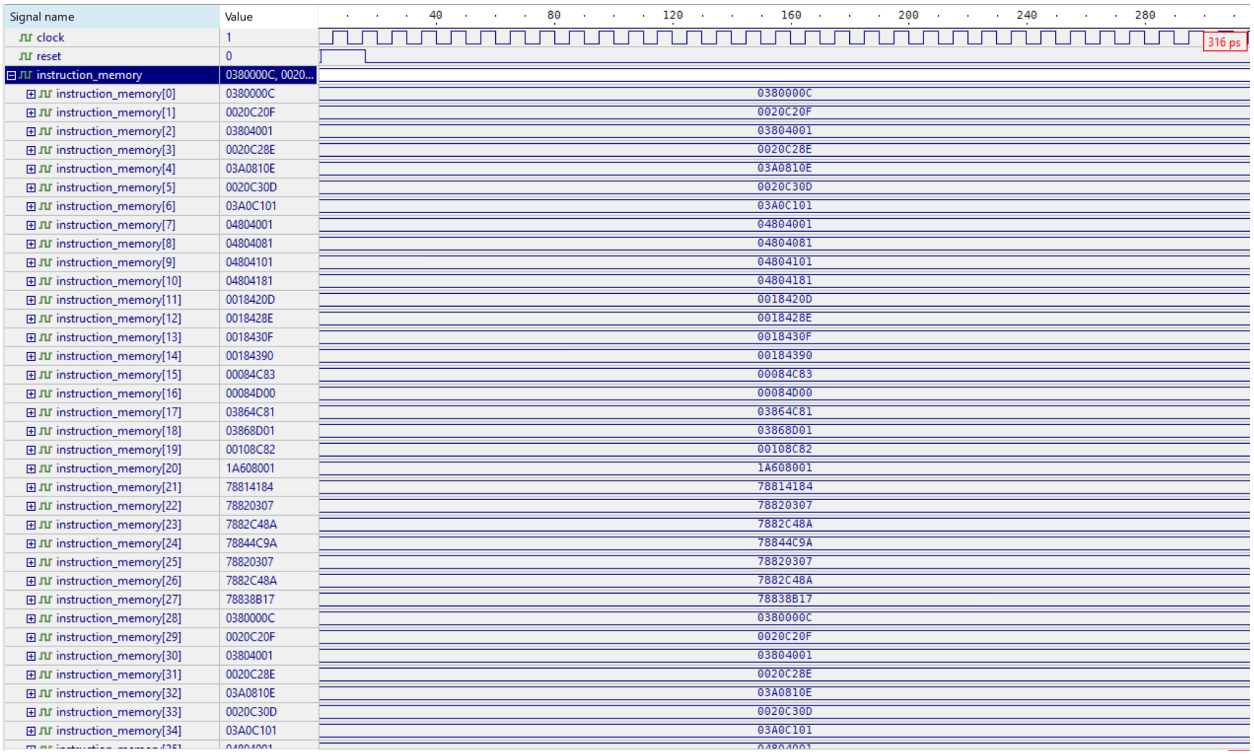
Instruction Fetch Test Bench Code:

```
/*************************************************************************************************************
 * Module: Instruction Fetch Testbench
 * Author: Noah Merone
 *-------------------------------------------------------------------------------------------------------------
 * Description:
 *     This testbench module verifies the functionality of the Instruction Fetch module by providing stimulus
 *     and monitoring the outputs.
 *-------------------------------------------------------------------------------------------------------------
 * Inputs:
 *   - clock: Clock signal
 *   - reset: Reset signal
 *   - ins_cache: Instruction cache containing 64B instructions
 *   - branch_taken: Signal indicating if a branch was taken
 *-------------------------------------------------------------------------------------------------------------
 * Outputs:
 *   - read_enable: Signal indicating if the Instruction Fetch unit is ready to read the next set of instructions
 *   - program_counter: Current program counter value
 *-------------------------------------------------------------------------------------------------------------
 * Internal Signals:
 *   - instr_d: Array storing the fetched instructions
 *   - stall: Signal indicating if the Instruction Fetch unit should stall fetching new instructions
 *   - program_counter_wb: Program counter to be used as the reset program_counter signal for Instruction Fetch to start reading new instructions
 *   - program_counter_check: Integer used for checkpointing to adjust the program counter while reading from ins_cache
 *************************************************************************************************************/

module Instruction_Fetch_TB ();

  logic clock, reset;
  logic [0:31] instruction_memory[0:255];
  logic [0:31] instruction[0:255];
  logic read_enable, stall;
  logic [7:0] program_counter;

  Instruction_Fetch instructions_fetch (
      clock,
      reset,
      instruction,
```

```
    program_counter,
    read_enable
  );

  initial clock = 0;

  always begin
    #5 clock = ~clock;
  end
  initial begin
    $readmemb("./Compiler/memory_contents.txt", instruction_memory);
    #1;
    reset = 1;
    @(posedge clock);
    #1;
    reset = 1;
    @(posedge clock);
    #1;
    reset = 0;
    // Pause the simulation for 100 time units, then stop the simulation (Stop and Signal)
    #300;
    $stop;
  end
  always @(posedge clock) begin
    if (read_enable == 1) begin
      instruction[0:255] = instruction_memory[(program_counter)+:256];
    end
  end
endmodule
```

Instruction Fetch Waveform:





Compiler:

Assembler:

The “assembler.py” script is created to function as an assembler, converting the assembly code instructions into their corresponding machine code representation. By importing the argparse module, it facilitates command-line argument parsing.

The main functionality of the script is in the ‘Assembler’ class, which is responsible for parsing the input assembly code, mapping assembly mnemonics to their respective opcodes, and then generating the corresponding machine code instructions.

1. Constructor (__init__):
- Initializes the assembler with the provided instruction list file name, source file path, destination file path, and debug mode status.
- Imports the instruction to opcode mapping from the specified instruction list file.
2. import_map method:
- Reads the instruction list file line by line, extracting operation mnemonics and their corresponding opcodes.
- Populates a dictionary (instruction_to_opcode_dict) with the mnemonic-opcode mappings.
3. interpret_line method:
- Parses a line of text from the input assembly file, ignoring comments and extraneous whitespace.
4. interpret_input method:
- Processes each line of the input assembly file, converting assembly mnemonics to machine code instructions.
- Writes the generated machine code instructions to the output destination file.
- Optionally logs debug information to a separate file based on the specified debug mode.
5. calculate method:
- Generates the binary representation of the machine code instruction based on the opcode and operands.
- Handles special cases such as memory offsets.
6. populate method:
- Ensures that binary sequences are padded to a specified span with leading zeros.

Finally, the script checks if its being run directly (__name__ == "__main__"), and if so it will use the argparse module to parse command line arguments. It then initializes the instance of the ‘Assembler’ class with the provided arguments and turns on the ‘interpret_input’ method to perform the assembly process.

Assembler Code:

```
import argparse

# python3.11 assembler.py input1.asm memory_contents.txt

class Assembler:
    def __init__(self, instruction_list_name, source_file, destination_file, debug):
        self.instruction_to_opcode_dict = {}
        self.source_file = source_file
        self.destination_file = destination_file
        self.log_severity = debug
        self.import_map(instruction_list_name)

    def import_map(self, instruction_list_name):
        with open(instruction_list_name, 'r') as instruction_list:
            for text_line in instruction_list:
```

```
            operation, operation_code = text_line.strip().split("\t")
            mnemonic = operation.split(" ")[0]
            operation_code = operation_code.strip()
            self.instruction_to_opcode_dict[mnemonic] = operation_code

    def interpret_line(self, text_line):
        return text_line.strip().split("//")[0].strip().split(" ")

    def interpret_input(self):
        with open(self.source_file, 'r') as instruction_list, \
                open(self.destination_file, 'w') as output_file, \
                open("debug.out", 'w') as data_object:
            for text_line in instruction_list:
                operation = self.interpret_line(text_line)
                mnemonic = operation[0]
                if "stop" in mnemonic:
                    break
                operation_code = self.instruction_to_opcode_dict.get(mnemonic, '00000000000')
                binary = self.calculate(operation_code, operation)
                output_file.write(binary + "\n")
                if self.log_severity == 2:
                    o_hex = '0x{0:0{1}X}'.format(int(binary, 2), 8)
                    data_object.write(str(binary) + "\t" + o_hex + "\t" + text_line)
                else:
                    data_object.write(str(binary) + "\t" + text_line)

    def calculate(self, operation_code, operation):
        instruction_binary = operation_code
        for op in operation[1:]:
            if '(' in op and ')' in op:
                reg_offset = op.split('(')[0]
                offset = op.split('(')[1].rstrip(')')
                reg_binary = self.populate(bin(int(reg_offset)).replace("0b", ""), 7)
                offset_binary = self.populate(bin(int(offset)).replace("0b", ""), 7)
                instruction_binary += reg_binary + offset_binary
            else:
                op_binary = self.populate(bin(int(op)).replace("0b", ""), 7)
                instruction_binary += op_binary
        return instruction_binary.zfill(32)


    def populate(self, sequence, span):
        sequence = sequence.lstrip('-').zfill(span)
        return '1' + sequence[1:] if sequence.startswith('-') else sequence
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Assembler script')
    parser.add_argument('source_file', help='Input file path')
    parser.add_argument('destination_file', help='Output file path')
    parser.add_argument('--debug', action='store_true', help='Enable debug mode')
    parser.add_argument('--instruction_list_name', default='instruction_list.txt', help='Instruction list file path')
    args = parser.parse_args()
    assembler = Assembler(args.instruction_list_name, args.source_file, args.destination_file, args.debug)
    assembler.interpret_input()
```

Instructions List:

fma rt, ra, rb, rc	1110
fnms rt, ra, rb, rc	1101

fms rt, ra, rb, rc 1111
ila rt, imm18 0100001
addx rt, ra, rb 01101000000
cg rt, ra, rb 00011000010
bg rt, ra, rb 00001000010
sfx rt, ra, rb 01101000001
ilh rt, imm16 010000011
il rt, imm16 010000001
ilhu rt, imm16 010000010
iohl rt, imm16 011000001
mpyi rt, ra, imm10 01110100
mpyui rt, ra, imm10 01110101
ahi rt, ra, imm10 00011101
ai rt, ra, imm10 00011100
sfhi rt, ra, imm10 00001101
sfi rt, ra, imm10 00001100
ceqhi rt, ra, imm10 01111101
ceqi rt, ra, imm10 01111100
cgthi rt, ra, imm10 01001101
cgti rt, ra, imm10 01001100
clgtbi rt, ra, imm10 01011110
clgthi rt, ra, imm10 01011101
clgti rt, ra, imm10 01011100
mpy rt, ra, rb 01111000100
mpya rt, ra, rb, rc 1100
mpys rt, ra, rb 01111000111
mpyhh rt, ra, rb 01111000110
mpyu rt, ra, rb 01111001100
mpyh rt, ra, rb 01111000101
fa rt, ra, rb 01011000100
fs rt, ra, rb 01011000101
fm rt, ra, rb 01011000110
shlh rt, ra, rb 00001011111
shl rt, ra, rb 00001011011
roth rt, ra, rb 00001011100
rot rt, ra, rb 00001011000
cntb rt, ra 01010110100
avgb rt, ra, rb 00011010011
absdb rt, ra, rb 00001010011
sumb rt, ra, rb 01001010011
ah rt, ra, rb 00011001000
a rt, ra, rb 00011000000
sfh rt, ra, rb 00001001000
sf rt, ra, rb 00001000000
and rt, ra, rb 00011000001
or rt, ra, rb 00001000001
xor rt, ra, rb 01001000001
nand rt, ra, rb 00011001001
ceqh rt, ra, rb 01111001000
ceq rt, ra, rb 01111000000
cgth rt, ra, rb 01001001000
cgt rt, ra, rb 01001000000
clgtb rt, ra, rb 01011010000
clgth rt, ra, rb 01011001000
clgt rt, ra, rb 01011000000
nop 01000000001
lnop 00000000001
shli rt, ra, imm7 00001111011
shlhi rt, ra, imm7 00001111111
rothi rt, ra, imm7 00001111100
roti rt, ra, imm7 00001111000

lqd rt, imm10(ra) 00110100
stqd rt, imm10(ra) 00100100
lqa rt, imm16 001100001
stqa rt, imm16 001000001
lqr rt, imm16 001100111
stqr rt, imm16 001000111
br imm16 001100100
bra imm16 001100000
binz rt, ra 00100101001
bihnz rt, ra 00100101011
brhz rt, imm16 001000100
brasl rt, imm16 001100010
brsl rt, imm16 001100110
brnz rt, imm16 001000010
brhnz rt, imm16 001000110
brz rt, imm16 001000000
shlqbi rt, ra, rb 00111011011
shlqby rt, ra, rb 00111011111
rotqby rt, ra, rb 00111011100
lqx rt, ra, rb 00111000100
stqx rt, ra, rb 00101000100
bi ra 00110101000
biz rt, ra 00100101000
bihz rt, ra 00100101010
shlqbi rt, ra, rb 00111011011
shlqbii rt, ra, imm7 00111111011
shlqbybi rt, ra, rb 00111001111
rotqbybi rt, ra, rb 00111001100
shlqbyi rt, ra, imm7 00111111111
rotqbyi rt, ra, imm7 00111111100
stop 00000000000