First Uber Cool Design

Register file, ALU, and Fetch unit

By: Brandon Chin (A11845998), Leandro Lubrico (A12077909), and Kevin Chen (A10623152)

Q1. Introduction

In this lab we have given substance to the First Uber Cool Design ISA. At the beginning of this lab, we had our goals set on making our code simple, as modular as possible, and easy to "hook together" in the future. We feel like we accomplished these goals. Our register file is simple. The ALU is complex, but modular enough where adding any instructions should be relatively simple. And the fetch unit robust and self-contained. We supported every instruction we had planned and the original design decisions seem to have held against the test of reality. Hopefully, while reading our report and supporting material, you will agree.

Q2. ISA Summary

Instruction Formats

Name	Format
A	[OP] [FUNC / SRC REG 1] [SRC REG 2 / IMMEDIATE] [3] [3] [3] Ex: load $r0, $r1 000 000 001
B	[OPCODE] [FUNC] [OFFSET] [3] [1] [5] Ex: bno if1 111 0 00101

Instructions

Name	OP / FUNC	Format	Operation
Load (load)	000	A	R[$rs1] = M[R[$rs2]]
Store (store)	001	A	M[R[$rs2]] = R[$rs1]
Add (add)	010	A	R[$rs1] = R[$rs1] + R[$rs2]
Match (match)	011	A	$ov = (R[$rs1] == R[$rs2])
Less than (lt)	100	A	if(R[$rs1] < R[$rs2]) $ov = 1, else $ov = 0
Distance (dist)	101	A	R[$rs1] = abs(R[$rs1] - R[$rs2])

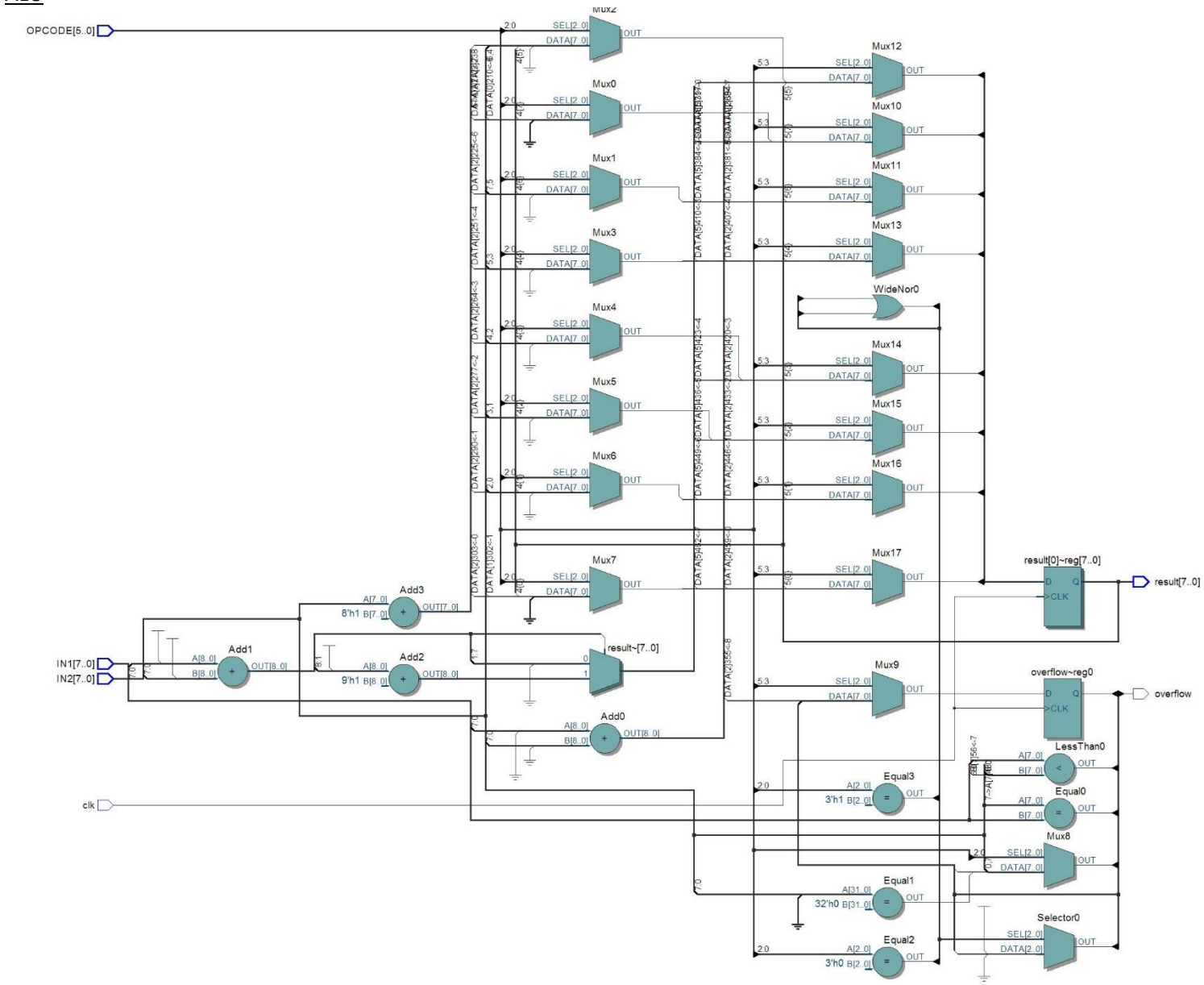
Logical Shift Left (lsl)	110 / 000	A	R[$rs] = R[$rs] << 1
Logical Shift Right (lsr)	110 / 001	A	R[$rs] = R[$rs] >> 1
Increment (incr)	110 / 010	A	R[$rs] = R[$rs] + 1
And with 1 (and)	110 / 011	A	$ov = LSB(R[$rs]) & 1
Equals 0 (eqz)	110 / 100	A	$ov = R[$rs] == 0
Assign 0 (zero)	110 / 101	A	R[$rs] = 0
TBD	110 / 110	A	
Halt (halt)	110 / 111	A	
Branch on No Overflow (bno)	111 / 0	B	if($ov == 0) PC = PC + SignedOffset;
Branch on Overflow (bof)	111 / 1	B	if($ov == 1) PC = PC + SignedOffset;

Q3. ALU Operations demonstration

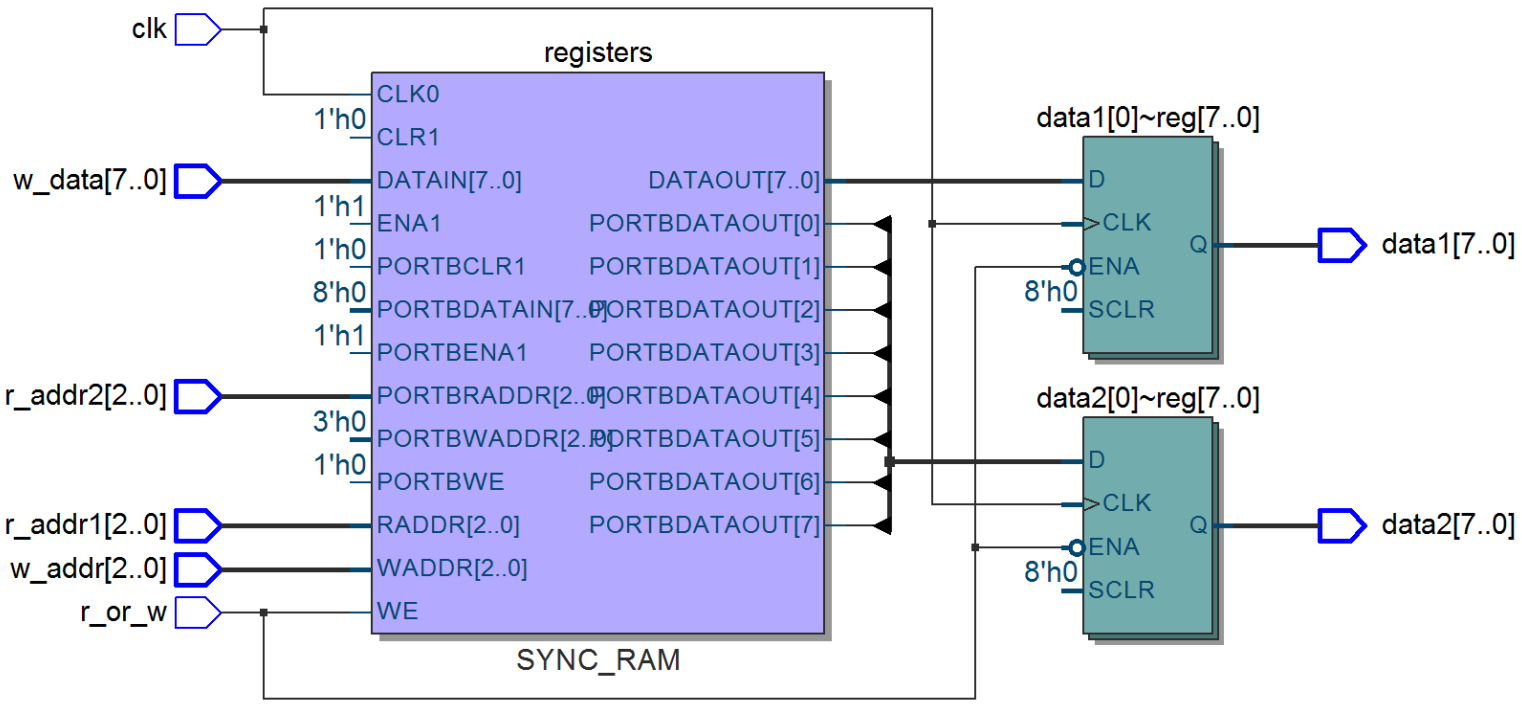
Add (add)	010	A	R[$rs1] = R[$rs1] + R[$rs2]
Match (match)	011	A	$ov = (R[$rs1] == R[$rs2])
Less than (lt)	100	A	if(R[$rs1] < R[$rs2]) $ov = 1, else $ov = 0
Distance (dist)	101	A	R[$rs1] = abs(R[$rs1] - R[$rs2])
Logical Shift Left (lsl)	110 / 000	A	R[$rs] = R[$rs] << 1
Logical Shift Right (lsr)	110 / 001	A	R[$rs] = R[$rs] >> 1
Increment (incr)	110 / 010	A	R[$rs] = R[$rs] + 1
And with 1 (and)	110 / 011	A	$ov = LSB(R[$rs]) & 1
Equals 0 (eqz)	110 / 100	A	$ov = R[$rs] == 0
Assign 0 (zero)	110 / 101	A	R[$rs] = 0
Branch on No Overflow (bno)	111 / 0	B	if($ov == 0) PC = PC + SignedOffset;
Branch on Overflow (bof)	111 / 1	B	if($ov == 1) PC = PC + SignedOffset;

Q4. Schematics and Verilog models

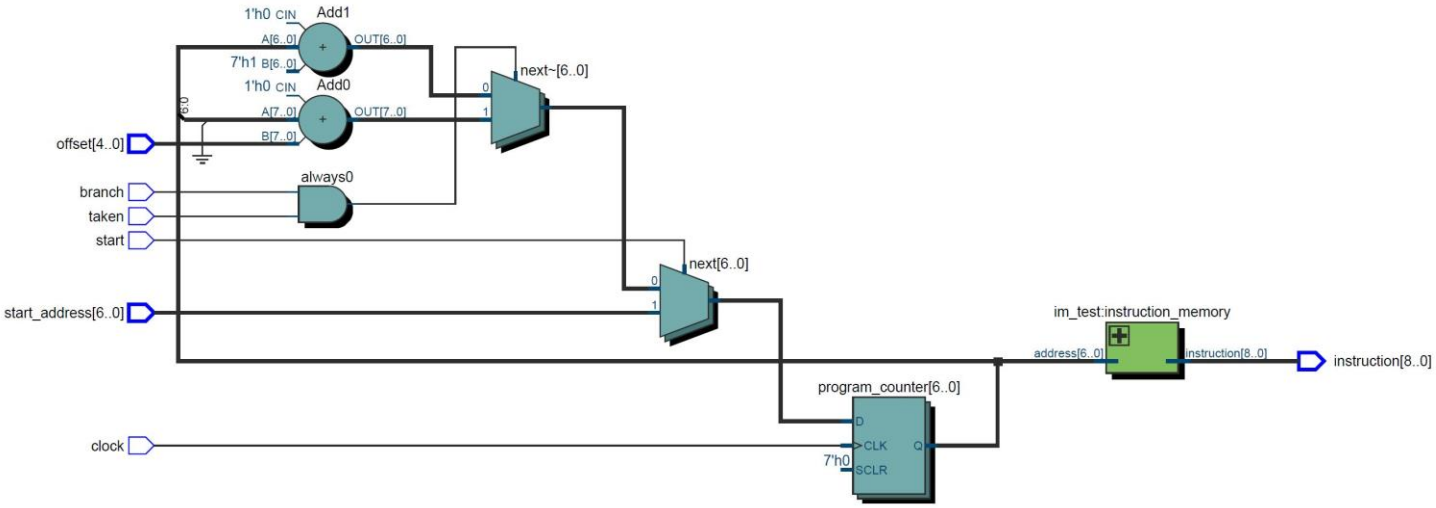
ALU



Register File

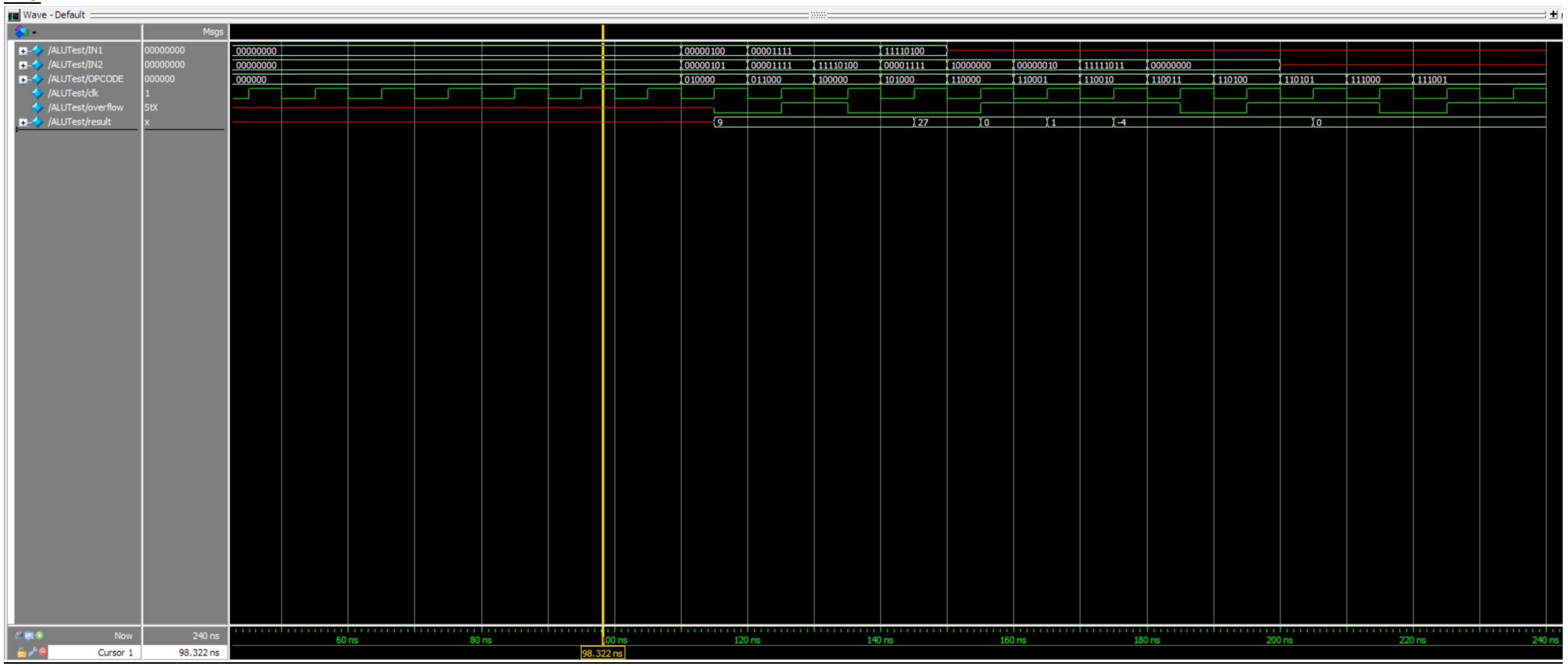


Fetch Unit



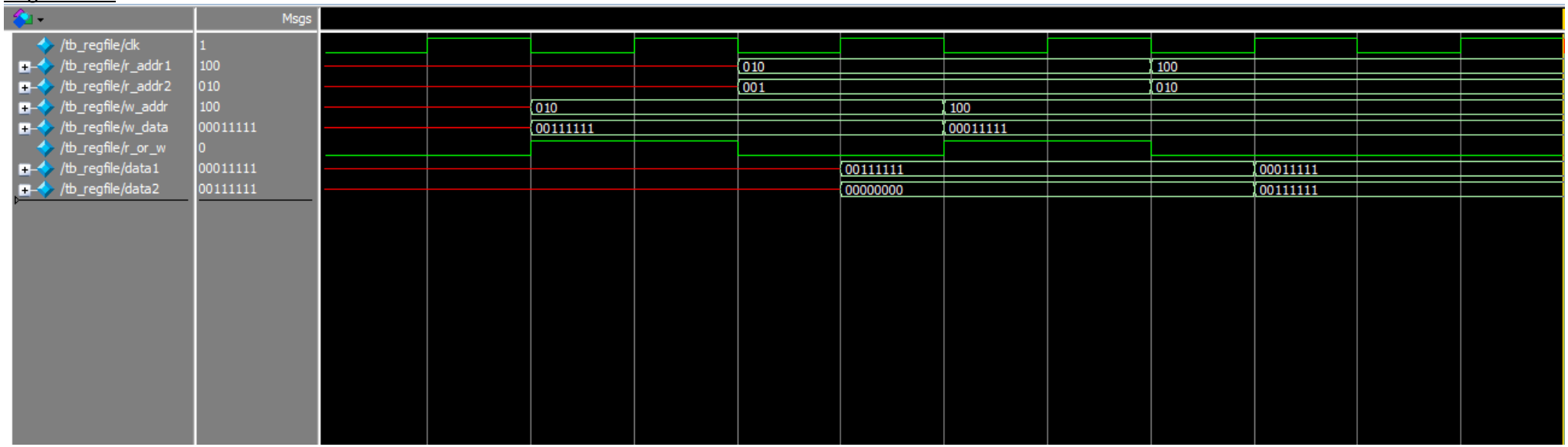
Q5. Timing diagrams

ALU



Opcode	Add (010000)	Match (011000)	Less Than (100000)	Dist (101000)	LSL (110000)	LSR (110001)	Increment (110010)	And1 (110011)	Equals 0 (110100)	Zero (110101)	Branch no overflow (111000)	Branch on overflow (111001)
IN1	4	00001111	15	-12	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care
IN2	5	00001111	-12	15	10000000	00000010	-5	00000000	00000000	Don't care	Don't care	Don't care
Result	9	Prev Val	Prev val	27	00000000	00000001	-4	Prev val	Prev val	0	Prev val	Prev val
Overflow	0	1	0	Prev val	1	Prev val	Prev val	0	1	Prev val	0	1

Register File



Register file naming legend

Name	r_addr1	r_addr2	w_addr	w_data	r_or_w	data1	data2
Meaning	Read Address 1	Read Address 2	Write Address	Data to write	Read or Write flag. 0 means read, while 1 means write	Output data stored at register index r_addr1	Output data stored at register index r_addr2

Annotation

In our register file, our registers are represented as eight 8-bit registers which all have initial values of 0, for now.

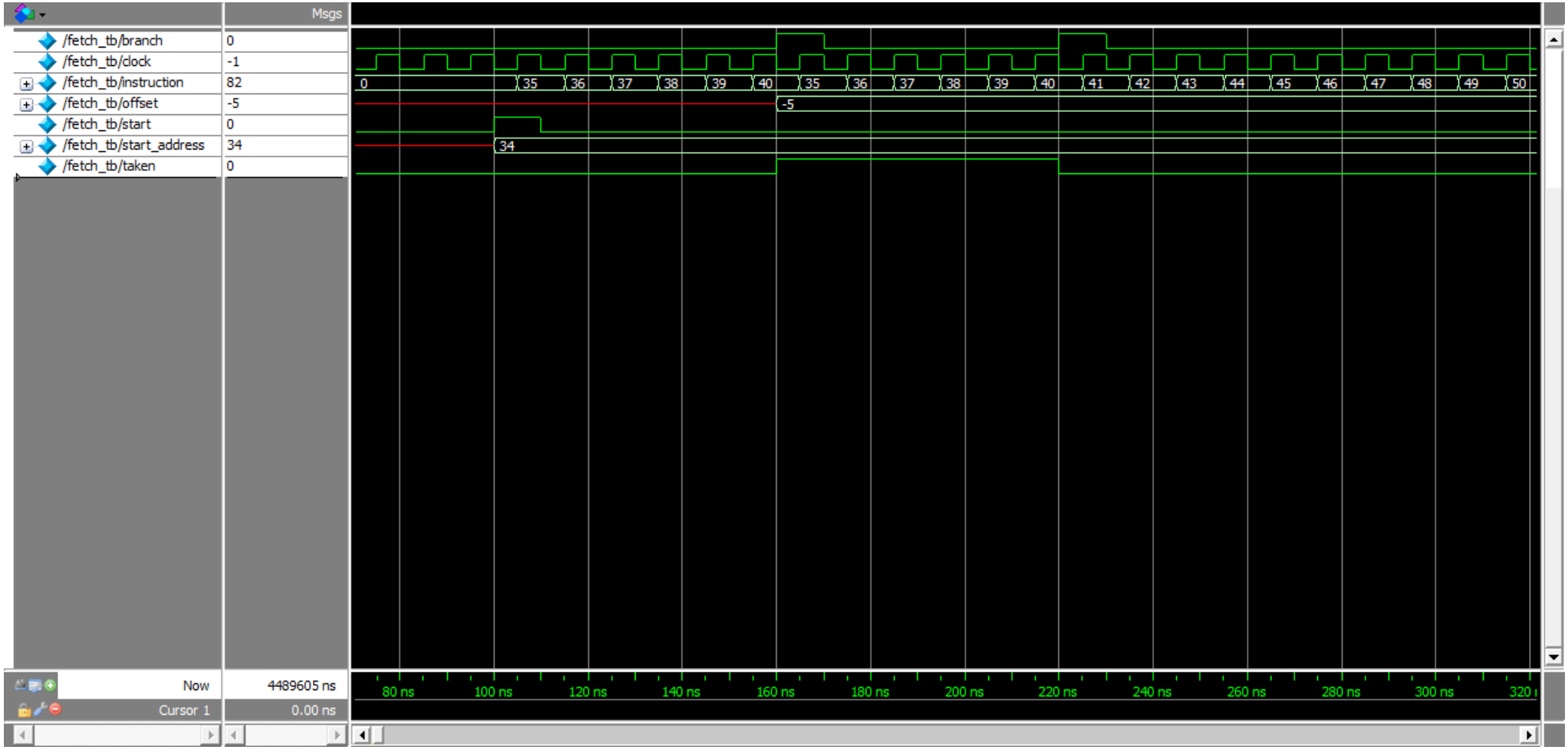
So in the first test case, we set w_addr1 = 010 (2), w_data = 00111111 (63), and r_or_w = 1 so that the value 63 gets written into the 2nd register. You cannot see the value of the register until the next test case where you we read from the register that was written to.

In the second test case, we set r_addr1 = 010 (2), r_addr2 = 001 (1), r_or_w = 0 so we read from the same register that we wrote to in the first test case. So the expected value in data1 = 00111111 (63), since we set that value in the previous test case, and data2 = 0, because the register at index 1 is not set to anything.

In the third test case, we set w_addr1 = 100 (4), w_data = 00011111 (31), and r_or_w = 1 so that we write 31 to the register 4, and once again we will not see the value in the timing diagram until the next test case where we read the address to the output data.

In the fourth test case, we set r_addr1 = 100 (4), r_addr2 = 010 (2), r_or_w = 0 so that we read from the same register that we wrote to in the first test case. So the expected value in data1 = 00111111 (63), since that is the value we set in the first test case, and data2= 00011111 (31), which is the value we set in the previous test case.

Fetch Unit



100ns:	start is asserted and start_address is set to 34
105ns:	instruction 35 (at address 34) is fetched
110ns:	start is deasserted to continue fetching instructions in sequence
115ns-165ns:	instructions are fetched sequentially since start and branch are deasserted
160ns:	branch and taken are asserted with offset set to -5 (to branch backwards 5 instructions)
165ns:	instruction 35 is fetch since branch and taken are asserted
170ps:	branch is deasserted to continue fetching instructions in sequence
175s-225ps:	instructions are fetched sequentially since start and branch are deasserted
220ps:	branch is asserted and taken is deasserted to indicate a branch instruction not taken
230ps:	branch is deasserted to indicate non-branch instructions
225ps-end:	instructions are fetched in sequence since start and branch are deasserted

Q7. Will your ALU be used for non-arithmetic instructions (e.g., address calculation, branches)? If so, how does that complicate your design?

Our ALU will not be used in non-arithmetic calculations. We have shifted the responsibility of address calculation and branches onto the fetch unit.

Q8. Name one thing you could have done differently in your ISA design to make your ALU design easier.

For A-type instructions with function codes, we have made it so that the first input does not matter and therefore we can assign values quicker.
We also have a well-defined instruction format, which makes for simple case statements, that reduces the amount of complexity in checking the instruction bits.

Q9. Name one thing you could have done differently in your ISA design to make your register file design easier.

We could have made it easier to set the write register bit by grouping all the instructions that write to a register to a similar opcode so that it doesn't require that much checking.

Q10. What is your most complex instruction, from the standpoint of the ALU?

The distance operation because it does signed arithmetic, a ternary conditional, and, within that ternary, we are comparing a specific bit index with 1. The ternary conditional would make a mux within the mux choosing the operation and another logic gate to do the comparison.

Q11. Now that your ALU is designed, are there any instructions that would be particularly straightforward to add given the hardware that is already there?

It would be straightforward to add a subtract instruction because we already have an adder, so all we would need is an inversion of the one of the inputs. It would also be straightforward to implement a greater than or equal to instruction, since we would just invert the output from the less than instruction.

Q12. Is there anything you could have done in your ISA to make your fetch unit design job easier?

We could have made a generic addition instruction in our ALU to handle address calculations for incrementing the program counter for non-branching and branching situations. The reason this would have simplified our design is because the fetch unit currently has to have extra logic to check the overflow bit, which is set by the ALU anyways, and do the addition within the unit. So the logic and hardware are both complicated with this gap in our ISA.