CSE 141L Final Report

Zuo Yang, A16631720; Zhengyu Huang, A16358704

Academic Integrity

Your work will not be graded unless the signatures of all members of the group are present beneath the honor code.

To uphold academic integrity, students shall:

- Complete and submit academic work that is their own and that is an honest and fair representation of their knowledge and abilities at the time of submission.
- Know and follow the standards of CSE 141L and UCSD.

Please sign (type) your name(s) below the following statement:

I pledge to be fair to my classmates and instructors by completing all of my academic work with integrity. This means that I will respect the standards set by the instructor and institution, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance. I pledge to excel with integrity.

Zuo Yang
Zhengyu Huang

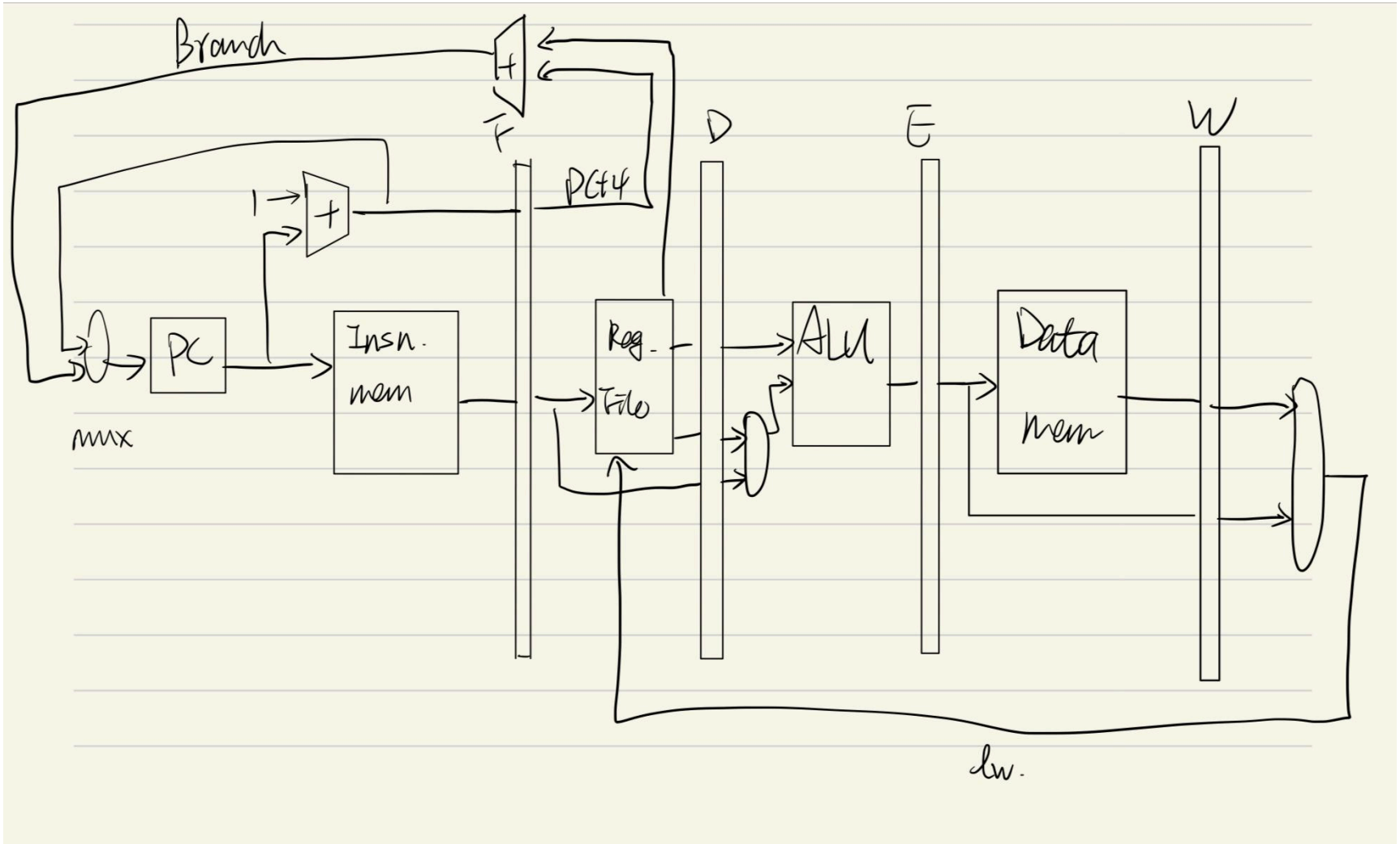
0. Team

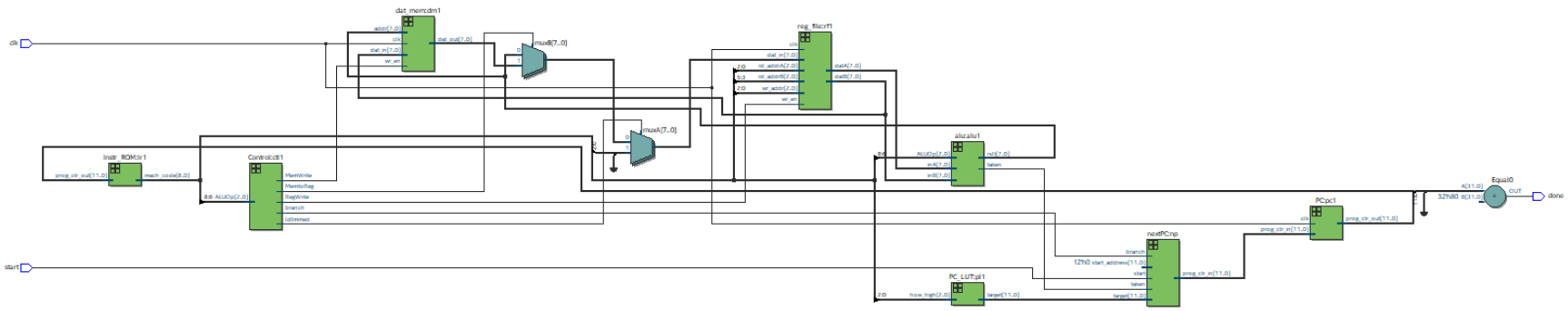
Zuo Yang, Zhengyu Huang

1. Introduction

Our architecture has a name of Low Bandwidth Deducer (LBD). The overall philosophy is to be clear and simple. The goal we have is to consume resources as little as possible. Our machine is a load-store machine. We load the desired operand from data mem into the reg file and perform bitwise operations and store the result back into data mem. Our goals include design of bitwise operations, load and store operations, branching operations, etc.

2. Architectural Overview




clk
start
dst_memrcdm1
muxB[7..0]
muxA[7..0]
reg_filecrf1
instr_ROMclr1
Controlctl1
alucalu1
nextPCcnp
PC_LUTpl1
PCpc1
Equal0
done

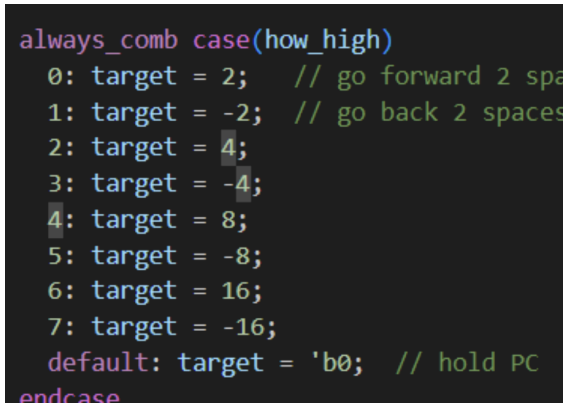
3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
2 source regs insns	3 bits opcode, 3 bits destination and source register, 3 bits source register	xor, beq, lw, sw, pos
1 source insn (3 bits reg)	3 bits opcode, 3 bits destination and source register, 3 bits intermediate	lrt
1 source isns (2 bits reg)	3 bits opcode, 2 bits destination register, 4 bits intermediate	ld

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
ld = load an intermediate to 3 LSB of the register	1 source regisns (2bits reg)	3 bits opcode (000), 3 bits source register(XXX), 3 bits intermediate (XXX)	ld R3 7⇔ `000_011_111` # after and instruction, R3 now holds `0b0000_0111`	Loading a 3 bits intermediate to the LSB of a register. Intermediate field can only take 0-7
rst = arithmetic right shift 1	1 source regs insns (3 bits reg)	3 bits opcode (001), 3 bits destination register (XXX)	# Assume R0 has `0b1010_0000` `lrt R0 101 ⇔ 001_000_101` # after and instruction, R0 now holds `0b1101_0000`	Right shift by 1
add = arithmetic add	2 source regs insns	3 bits opcode (010), 3 bits destination and source register (XXX), 3 bits source register(XXX)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` `add R0 R1 ⇔ 010_000_001` # after and instruction, R0 now holds `0b1010_0001`	
pos = positive	2 source regs	3 bits opcode (011), 3 bits destination and source register (XXX), 3 bits	# Assume R0 has `0b1001_0000` # Assume R1 has `0b0000_0001`	If R1 > 0, go to the memory location at R0 Pos compares if data in R1 is positive, if so, take the branch.

number	insns	source register(XXX)	`pos R0 R1 ⇔ 011_000_001` # after and instruction, the current address is saved in LUT, and PC is now `0b1001_0000`	
xor = bitwise xor	2 source regs insns	3 bits opcode (100), 3 bits destination and source register (XXX), 3 bits source register(XXX)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` `xor R0 R1 ⇔ 100_000_001` # after and instruction, R0 now holds `0b1000_0001`	The data of the first reg will be replaced If want to have “~r2”, can use the following: ld r1, 4b1111 lrt r1, 4 ld r1, 4b1111 xor r2, r1
Jmp = jump to if equals to 0	1 source reg insns	3 bits opcode (101), 3 bits source register (XXX), 3 bits jump code(XXX)	# Assume R0 has 0b0000_0000 # Assume jmp code is 6 = 0b110 # Assume pc = 0b1000_0000 beq R0 R1 ⇔ 101_000_110 # after and instruction, PC now equals to 0b1001_0000	
lw = load word	2 source regs insns	3 bits opcode (110), 3 bits destination register (XXX), 3 bits register for memory origin (XXX)	# Assume R1 has `0b1011_0011` # Assume memory 0xb3 has `0b1111_1111` `lw R0 R1 ⇔ 110_000_001`	0b10110011=0xb3

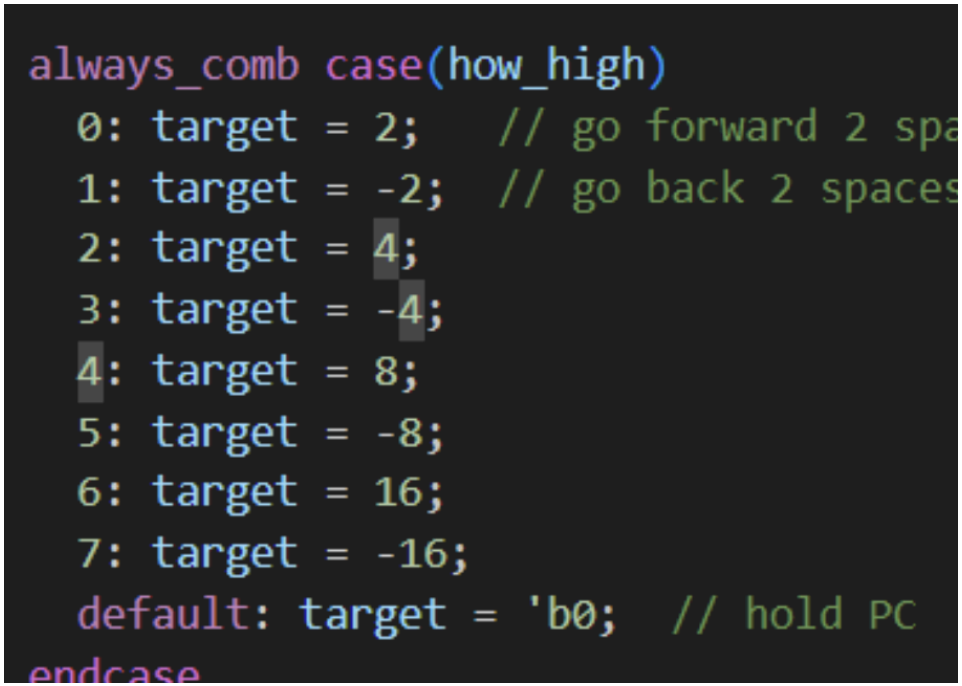
			# after and instruction, R0 now holds `0b1111_1111`	
sw = store word	2 source regs insns	3 bits opcode (111), 3 bits source register (XXX), 3 bits register for memory destination(XXX)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1011_0011` `sw R0 R1 ⇔ 111_000_001` # after and instruction, memory 0xb3 now holds `0b0001_0001`	

Internal Operands

5 general purpose registers storing 00000000 each. R5 stores constant 11111111. R6 stores constant 00000001. R7 stores constant 01000000. 1 program counter.

Control Flow (branches)

We will use jump and no-operation codes to control the flow. If register input of jmp has a value 0, modify pc according to the value from the look-up table by jump code input (see below). Use no–operations insns in between jumps gaps, (e.g. lrt R0 0).



```
always_comb case(how_high)
  0: target = 2;   // go forward 2 spa
  1: target = -2;  // go back 2 spaces
  2: target = 4;
  3: target = -4;
  4: target = 8;
  5: target = -8;
  6: target = 16;
  7: target = -16;
  default: target = 'b0;  // hold PC
endcase
```

Addressing Modes

Direct. Addr = reg. Only 256 addresses supported
E.g. to access mem@ 0xde
ld R0, 0xd
lrt R0, 4
ld R0, 0xe
lw R1, R0

4. Programmer's Model [Lite]

4.1 How should a programmer think about how your machine operates? Provide a description of the general strategy a programmer should use to write programs with your machine. For example, one could say that the programmer should prioritize loading in the necessary values from memory into as many registers as possible, then perform calculations. Another approach could be loading and writing to memory in between every calculation step. Word limit: 200 words.

Our machine requires the programmer to load everything from data memory to registers before executing programs, because our machine is a load-store machine and it uses absolute addressing. Load and store uses direct addressing. To perform a for loop, first store the value of index in one of the gp registers. Then use beq to store the absolute address of the for loop instruction in another gp register. After performing the instructions inside the for loop, increment the index, and load the address of for loop back to pc to complete the jump. The loop will terminate once the index is equal to the last value. For conditionals, to compare to values, first make one side negative. Then add the new value with the other one. Finally we use pos to check if the result is positive or negative.

4.2 Can we copy the instructions/operation from MIPS or ARM ISA? If no, explain why not? How did you overcome this or how do you deal with this in your current design? Word limit: 100 words.

No. there are a lot of bit length differences in the instructions for the bit budget purpose. We limit some of the insns like ld only accessible to a few registers.
We borrowed some of the ideas of instructions in MIPS ISA in our instructions design. Instructions like add, lrt, sw, lw, and xor. But our beq is different because it only compares the value to 0. Pos is our unique instruction.

4.3 Will your ALU be used for non-arithmetic instructions (e.g., MIPS or ARM-like memory address pointer calculations, PC relative branch computations, etc.)? If so, how does that complicate your design?

No

5. Individual Component Specification

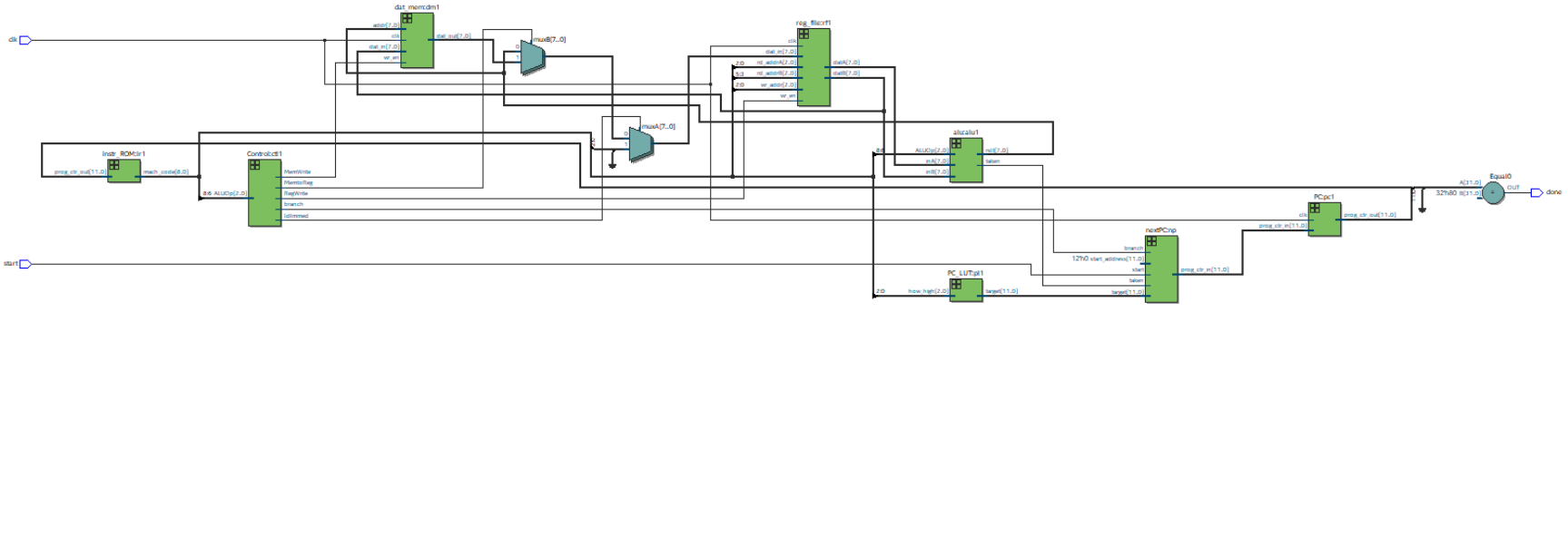
Top Level

Module file name: Top_level.sv

Functionality Description

The `top_level` module encapsulates the core functionality of a processor's architecture, integrating critical components such as the program counter, arithmetic logic unit (ALU), register file, instruction ROM, data memory, and control logic. Its primary function is to orchestrate the execution of machine code by managing the flow of data and control signals throughout the processor. This includes fetching and decoding instructions, executing arithmetic and logical operations via the ALU, handling data storage and retrieval through the data memory, and dynamically managing instruction flow with conditional and unconditional jumps. The module also utilizes multiplexers to route data effectively and controls various operational flags to manage register and memory operations. Designed to synchronize operations with the system clock and respond to start signals, the `top_level` module ensures that the processor efficiently processes tasks and meets operational benchmarks, signaling completion when the program counter reaches a predetermined end condition.

Schematic


clk
start
dat_memrdm1
muxB[7..0]
muxA[7..0]
reg_filerf1
instr_ROMirl
Controlctl1
alucallu1
PCpc1
nextPCnp
PC_LUTplt1
Equal0
done

Program Counter

Module file name: PC.sv
Module testbench file name: testbenches/PC/PC_testbench.sv

Functionality Description;

The `PC` (Program Counter) module serves a fundamental role in managing the sequence of instruction execution within a processor by maintaining the current instruction address. It is parameterized with `D=12`, which specifies that the program counter can handle a 12-bit wide address, suitable for indexing a moderate size of instruction memory. The module operates purely on a synchronous basis, updating the program counter only on the positive edge of the clock signal (`clk`).

This module takes a single input, `prog_ctr_in`, which is the next instruction address, either computed externally based on current execution needs such as conditional branches, loops, or direct jumps, and outputs it through `prog_ctr_out`. The simplicity of the module's functionality, where the output program counter simply latches the input address at each clock cycle, facilitates a reliable and straightforward incrementation or modification of the program sequence. This mechanism supports both relative and absolute jumps within the processor's control flow, allowing it to dynamically adjust execution paths efficiently, which is critical for implementing loops, conditional operations, and direct jumps within the program's logic.

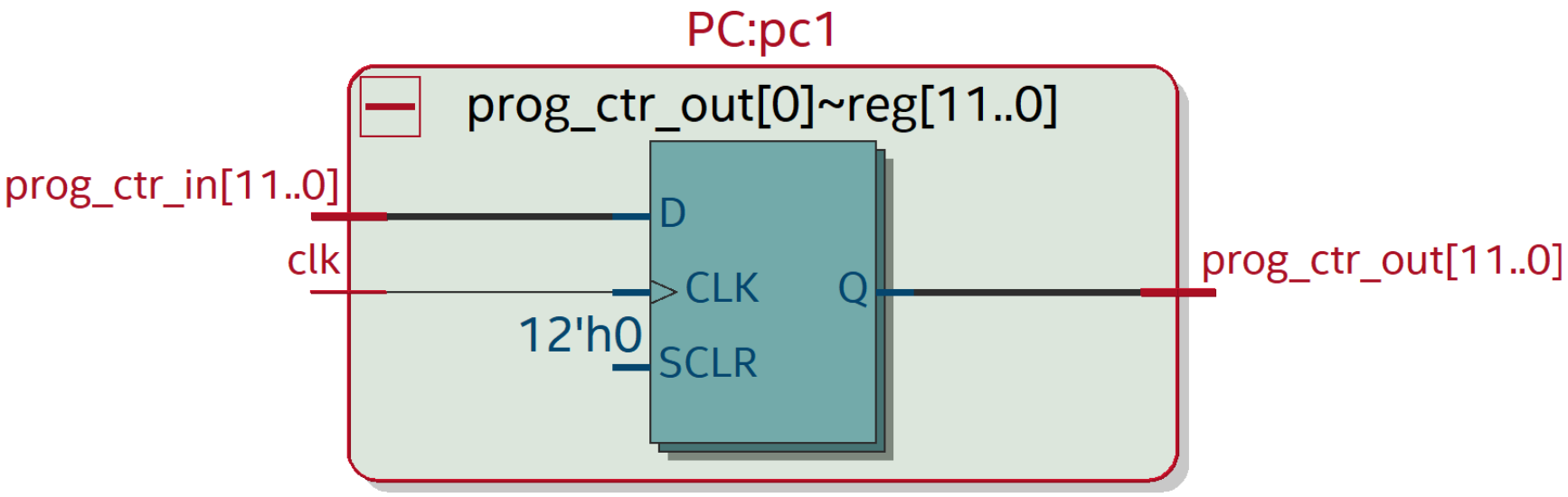
(Optional) Testbench Description

The test bench will instantiate a PC and a nextPC. Both will be run to test if the behavior is correct under different corresponding conditions.

Cases:

1. Initialize
2. Normal incrementing
3. Only *branch* signal
4. Only *taken* signal
5. *jump* 16 and -2
6. *start* at 128

Schematic


PC:pc1
prog_ctr_out[0]~reg[11..0]
prog_ctr_in[11..0]
clk
12'h0
D
CLK
SCLR
Q
prog_ctr_out[11..0]

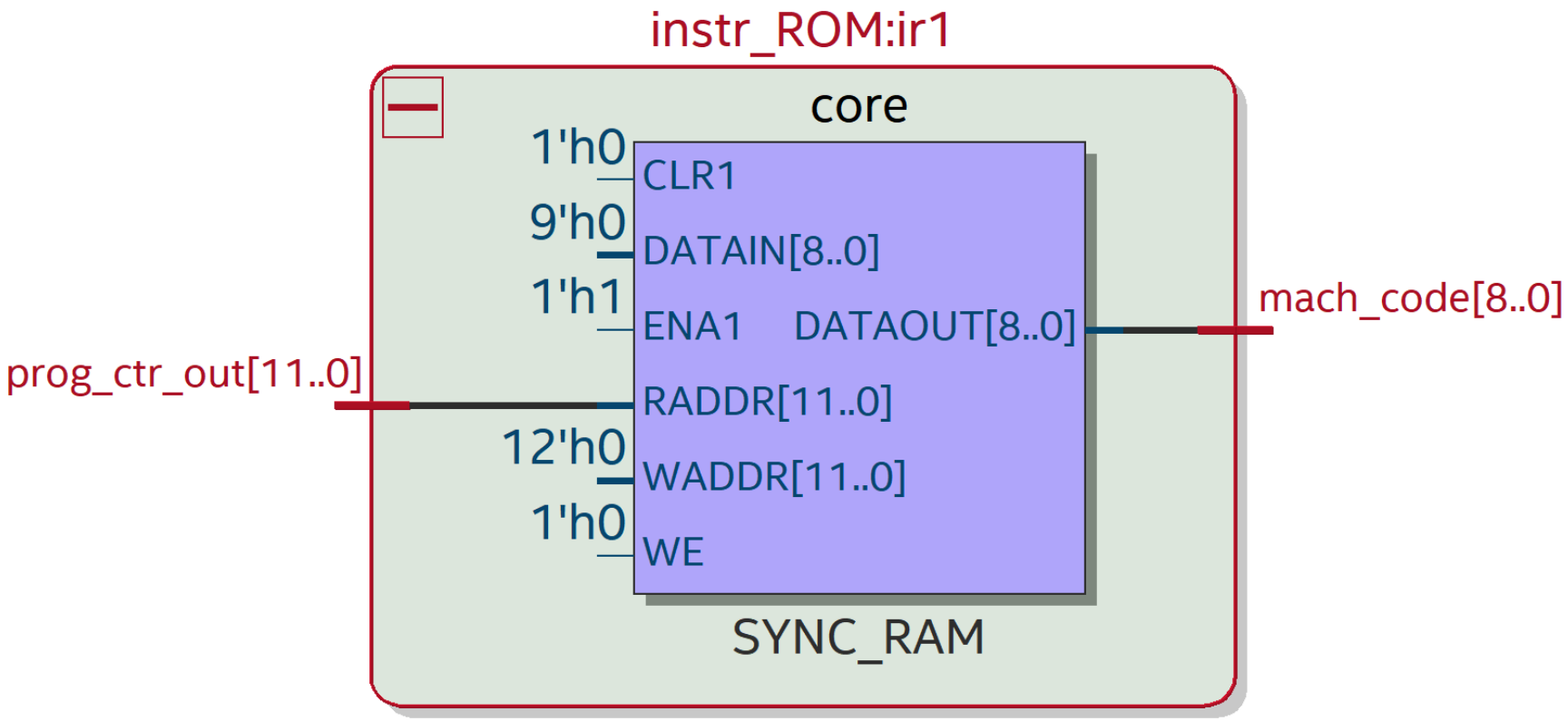
Instruction Memory

Module file name: instr_ROM.sv

Functionality Description

The instr_ROM module functions as a read-only memory (ROM) for storing and retrieving machine code instructions in a processor architecture. It is parameterized by D, which defines the width of the program counter, allowing for flexibility in addressing range. The module takes an input prog_ctr_out, which serves as the address pointer from the program counter, and outputs mach_code, a 9-bit wide instruction. The core of the module is a memory array core that holds the machine code instructions, loaded initially from an external file mach_code.txt. The module continuously outputs the instruction corresponding to the current program counter value, enabling the processor to fetch and execute the next operation.

Schematic


instr_ROM:ir1
core
1'h0
CLR1
9'h0
DATAIN[8..0]
1'h1
ENA1
DATAOUT[8..0]
mach_code[8..0]
prog_ctr_out[11..0]
RADDR[11..0]
12'h0
WADDR[11..0]
1'h0
WE
SYNC_RAM

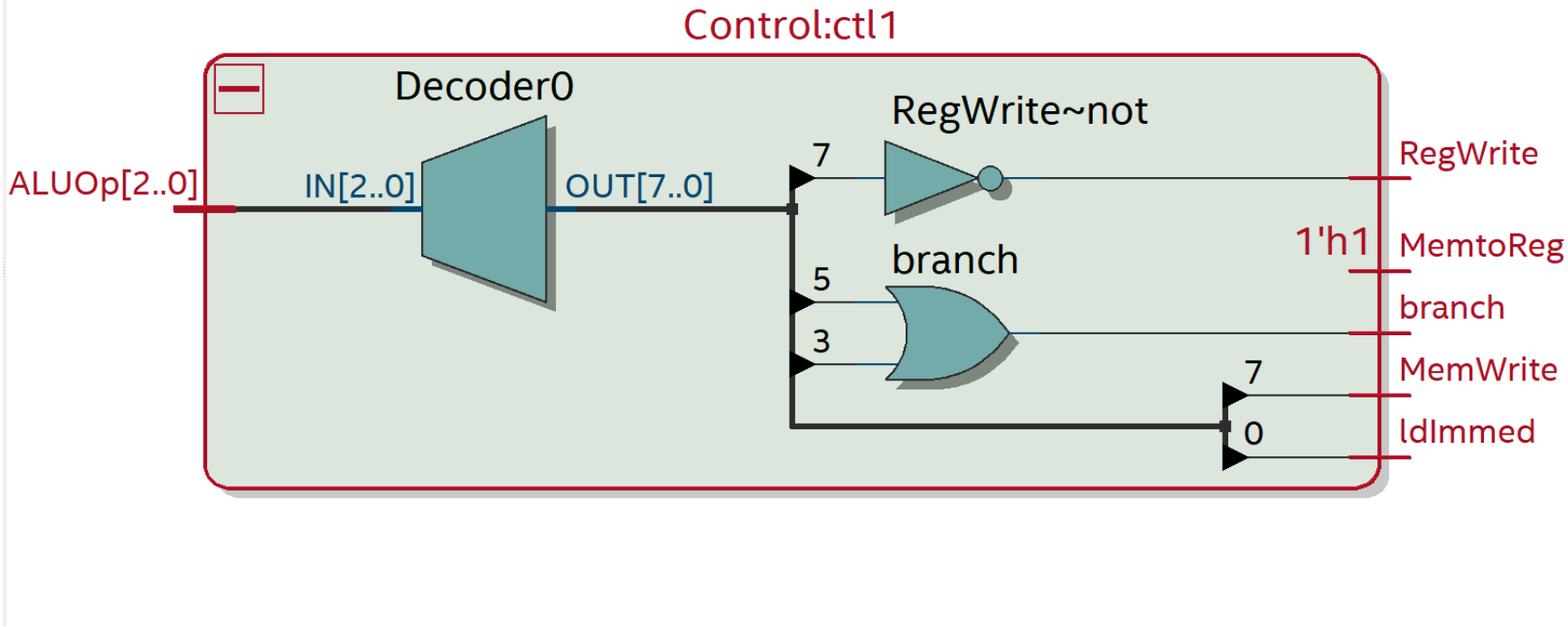
Control Decoder

Module file name: Control.sv

Functionality Description

Parameterized by `opwidth` to accommodate various opcode widths, this module inputs an opcode (`ALUOp`) and outputs control signals such as `branch`, `ldImmed`, `MemtoReg`, `MemWrite`, and `RegWrite`. Within its `always_comb` block, the module sets default signal states suitable for most operations but adjusts these based on specific opcode values. For example, it enables `ldImmed` for immediate loading, activates `branch` and disables `RegWrite` for jump and branch instructions, sets `MemtoReg` for load operations to route data from memory to registers, and enables `MemWrite` while disabling `RegWrite` for store operations. This dynamic and responsive decoding ensures precise control over data flow and operational execution within the processor, aligning each component's activity with the current instruction's requirements.

Schematic


Control:ctl1
Decoder0
ALUOp[2..0]
IN[2..0]
OUT[7..0]
7
RegWrite~not
RegWrite
1'h1
MemtoReg
5
branch
branch
3
7
MemWrite
0
ldImmed

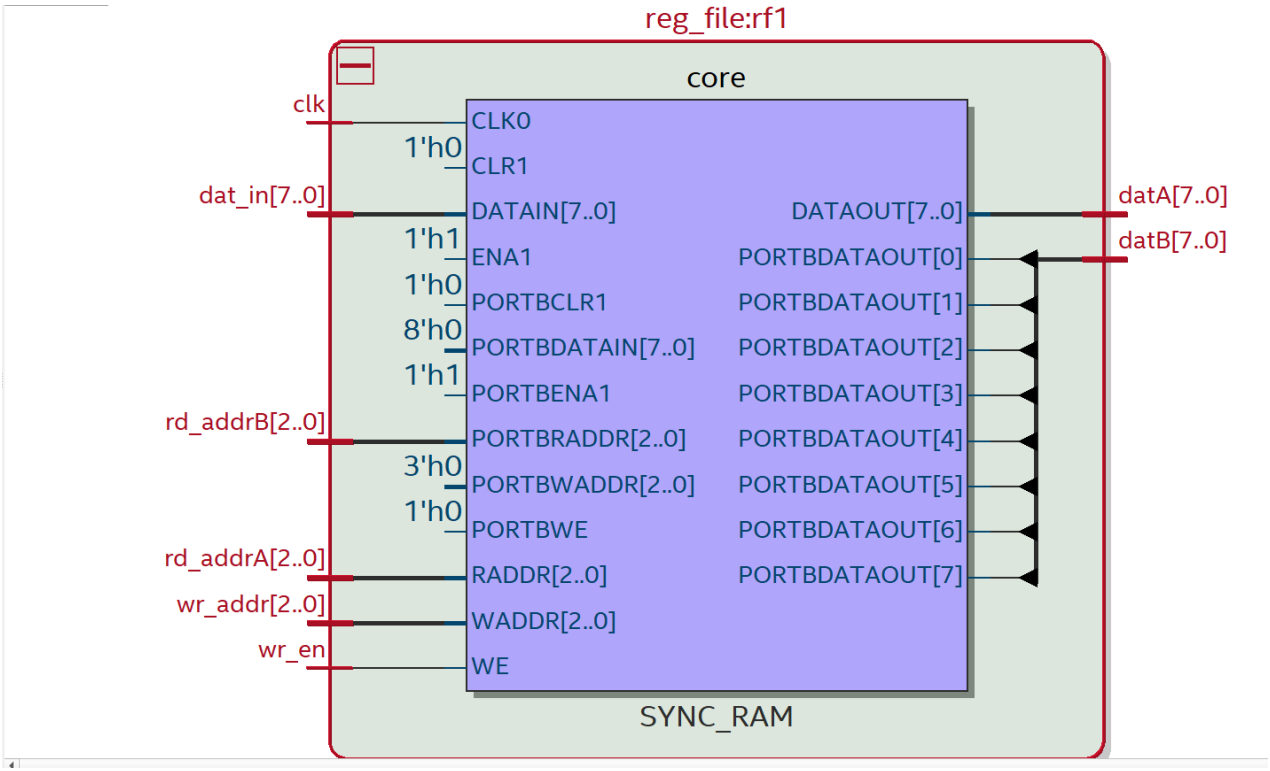
Register File

Module file name: reg_file.sv

Functionality Description

The `reg_file` module serves as a register file or cache memory within a processor, managing data storage and retrieval for a specified number of registers. It is parameterized with `pw`, which defines the pointer width and therefore the number of registers it can address, defaulting to manage 16 registers if `pw` is set to 4. This module takes inputs for data (`dat_in`), a clock signal (`clk`), a write enable signal (`wr_en`), and addresses for writing (`wr_addr`) and reading (`rd_addrA`, `rd_addrB`). It outputs data from two specified registers (`datA` and `datB`). Internally, it uses a 2-dimensional array `core` to store register data, initializing specific registers to predetermined values (e.g., `0`, `-1`, `1`, `64`) for default states or specific functional purposes. Read operations are handled combinationaly, providing immediate access to the register data via `datA` and `datB` based on the read addresses. Write operations are clocked and occur on the rising edge of the clock signal, updating the register specified by `wr_addr` with `dat_in` only when `wr_en` is asserted, ensuring controlled and timely data updates within the register file. This setup effectively supports concurrent read operations and synchronized writes, crucial for efficient and reliable processor operations.

Schematic


reg_file:rf1
core
clk
dat_in[7..0]
rd_addrB[2..0]
rd_addrA[2..0]
wr_addr[2..0]
wr_en
1'h0
1'h1
1'h0
8'h0
1'h1
3'h0
1'h0
CLK0
CLR1
DATAIN[7..0]
ENA1
PORTBCLR1
PORTBDATAIN[7..0]
PORTBENA1
PORTBRADDR[2..0]
PORTBWADDR[2..0]
PORTBWE
RADDR[2..0]
WADDR[2..0]
WE
DATAOUT[7..0]
PORTBDATAOUT[0]
PORTBDATAOUT[1]
PORTBDATAOUT[2]
PORTBDATAOUT[3]
PORTBDATAOUT[4]
PORTBDATAOUT[5]
PORTBDATAOUT[6]
PORTBDATAOUT[7]
datA[7..0]
datB[7..0]
SYNC_RAM

ALU (Arithmetic Logic Unit)

Module file name: alu.sv
Module testbench file name: alu_tb.sv

Functionality Description

The alu processes 8-bit wide data inputs, `inA` and `inB`, according to the operation code specified by `ALUOp`. The outputs include `rslt`, which stores the result of the operation, and `taken`, a flag used primarily for controlling flow within the processor. Operations supported by the ALU include direct data transfer (`ld`), arithmetic right shift, addition, bitwise XOR, and control operations such as checking if a number is positive (`pos`) and branch if equal (`beq`). These operations allow the ALU to perform essential computational tasks such as data manipulation, arithmetic calculations, and decision-making based on comparisons, which are critical for executing program logic and managing data flow within the system.

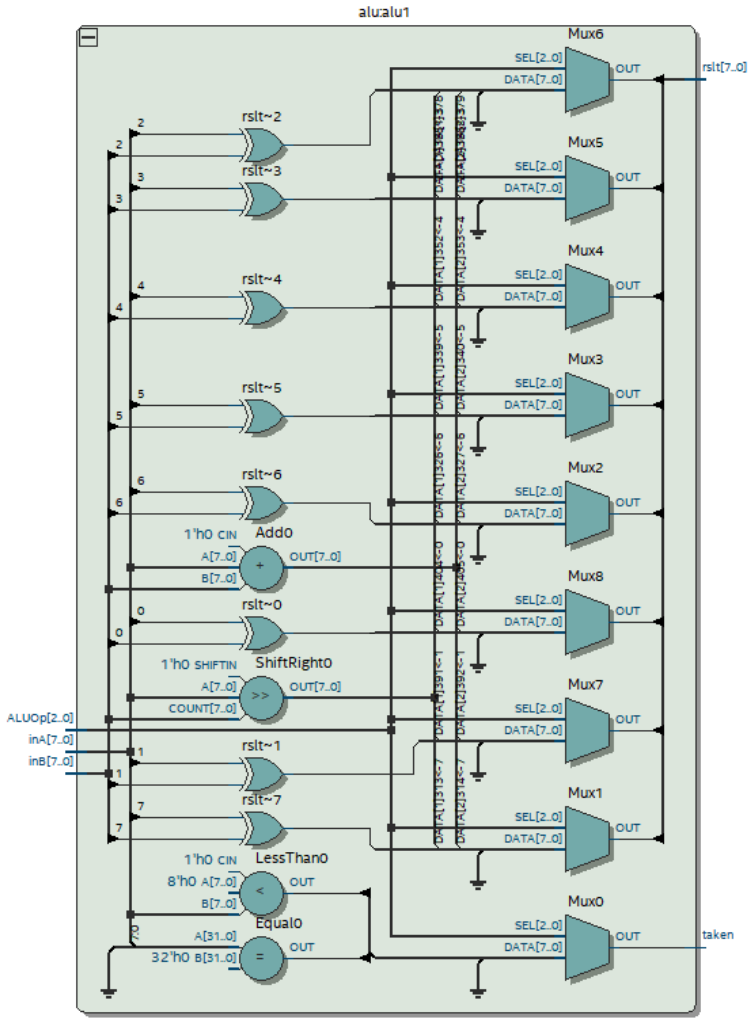
Testbench Description

The `alu_tb` testbench verifies the functionality of the ALU module by initializing inputs and applying a series of test cases with different ALU operations (`ALUOp`) and input values (`inA`, `inB`). It executes these test cases sequentially, with each case running for 10 nanoseconds, and monitors the outputs (`rslt` and `taken`). The test cases include various ALU operations to ensure the ALU performs correctly under different conditions. The `$monitor` task prints input and output values to provide real-time feedback on the ALU's behavior. Finally, the simulation is terminated using `$stop` after all test cases are executed, validating the ALU's accuracy and functionality.

ALU Operations

001: left shift
010: add
011: if positive number, taken is 1.
100: bitwise xor.
101: if equal to 0, taken is 1.

Schematic


alu:alu1
Mux6
Mux5
Mux4
Mux3
Mux2
Mux8
Mux7
Mux1
Mux0
SEL[2..0]
DATA[7..0]
OUT
rslt[7..0]
taken
rslt~2
rslt~3
rslt~4
rslt~5
rslt~6
rslt~0
rslt~1
rslt~7
1'h0 CIN
Add0
A[7..0]
B[7..0]
OUT[7..0]
1'h0 SHIFTIN
ShiftRight0
COUNT[7..0]
LessThan0
8'h0 A[7..0]
Equal0
A[31..0]
32'h0 B[31..0]
ALUOp[2..0]
inA[7..0]
inB[7..0]

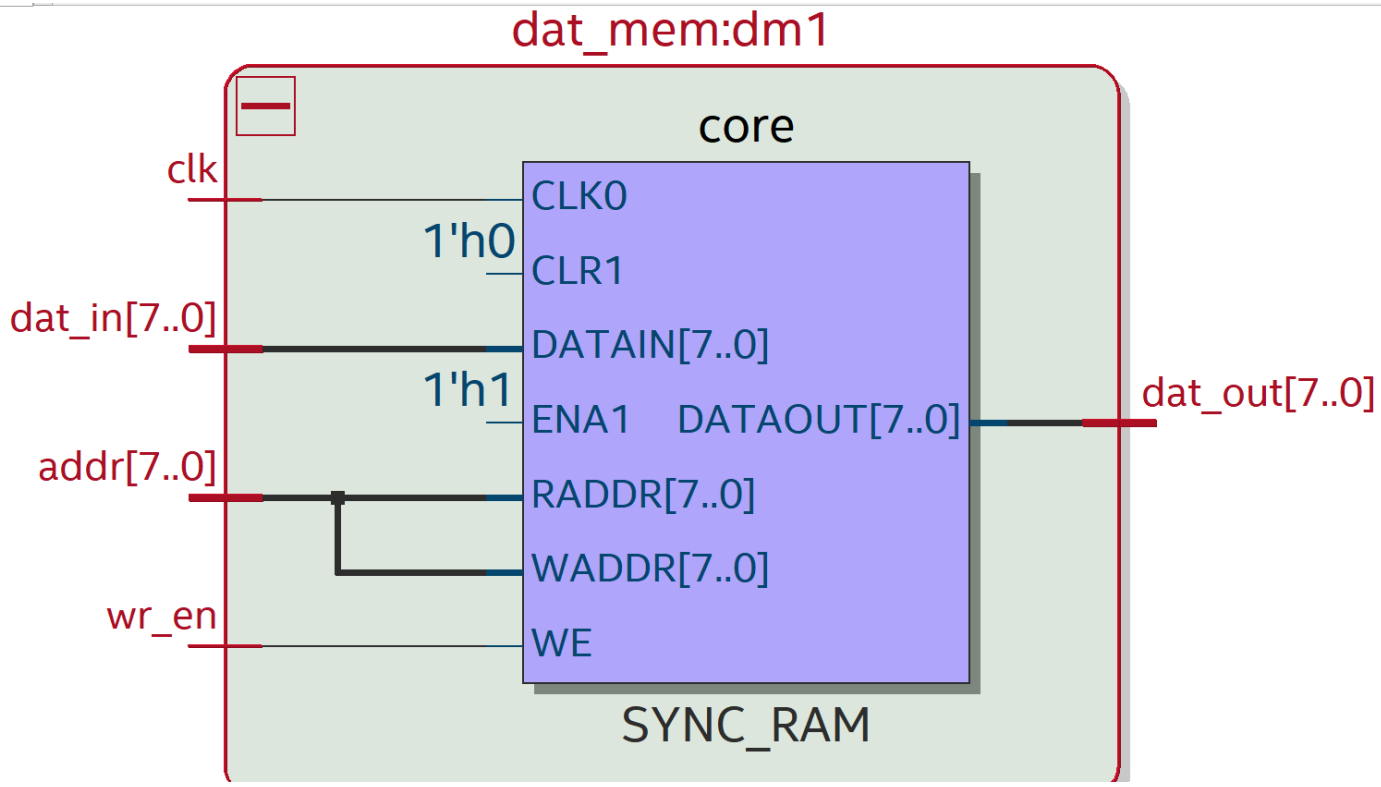
Data Memory

Module file name: dat_mem.sv

Functionality Description

The `dat_mem` module is designed as an 8-bit wide, 256-word deep memory array, providing essential data storage functionality in a digital system. This module handles both the reading and writing of data, with the data input (`dat_in`), memory address (`addr`), and clock signal (`clk`) facilitating these operations. Reads from this memory are combinational, allowing immediate access to data at any specified address without clock synchronization, which is output through `dat_out`. Writes, however, are controlled and occur only on the positive edge of the clock when the write enable (`wr_en`) signal is asserted. This distinction ensures data integrity by allowing updates to memory only during specific times, making the `dat_mem` module a reliable and efficient component for managing data storage in various computing environments, where quick data retrieval and secure data update mechanisms are crucial.

Schematic


dat_mem:dm1
core
clk
1'h0
dat_in[7..0]
1'h1
addr[7..0]
wr_en
CLK0
CLR1
DATAIN[7..0]
ENA1
DATAOUT[7..0]
RADDR[7..0]
WADDR[7..0]
WE
SYNC_RAM
dat_out[7..0]

Lookup Tables

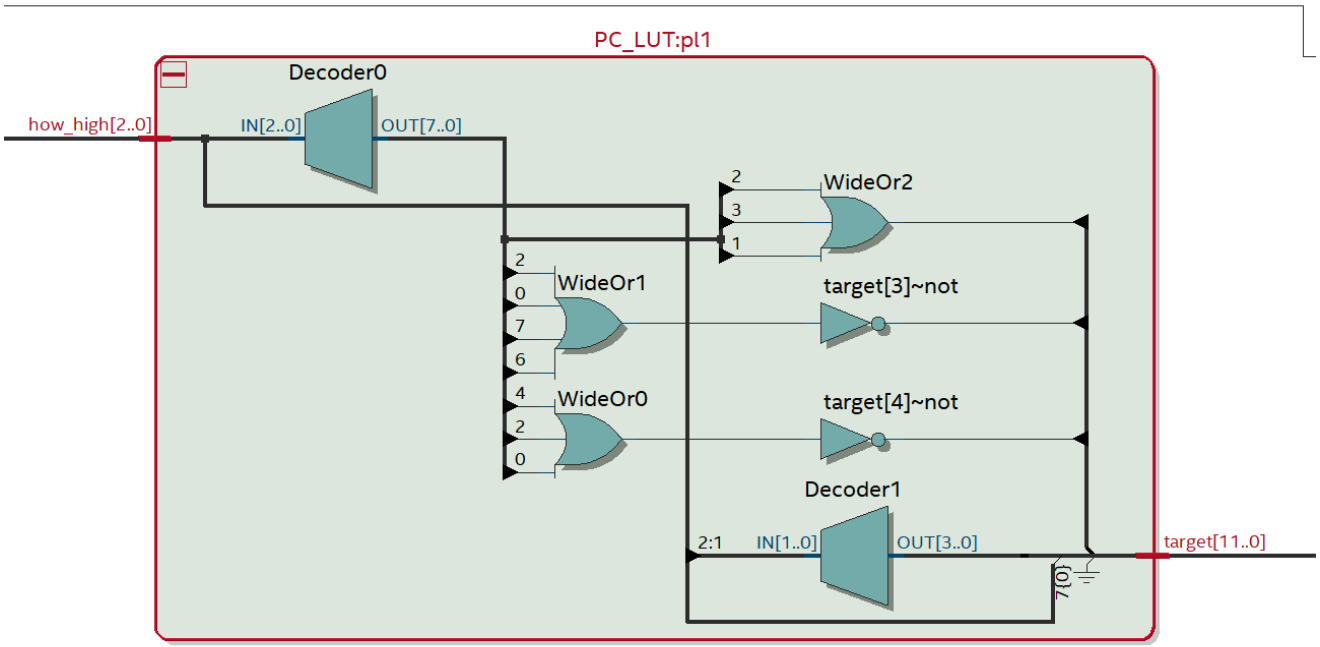
Module file name: PC_LUT.sv

Functionality Description

The `PC_LUT` (Program Counter Lookup Table) module is tailored to control the flow of a processor's program counter by providing specific target addresses based on input conditions. Parameterized by `D=12`, this module supports addressing in a 12-bit space, making it suitable for handling moderately sized address spaces. The module accepts a 3-bit input, `how_high`, which selects from a predefined set of target addresses for branching purposes within the program's execution. Based on the value of `how_high`, the module outputs a 12-bit `target` address which can be either positive or negative, facilitating both forward and backward jumps within the program.

The behavior of the module is defined using an `always_comb` block with a case statement, which assigns specific hard-coded target addresses for each possible input value. These targets range from small displacements like 2 or 8 to larger displacements for looping constructs, both positive (forward jump) and negative (backward jump). Default behavior is set to return `0`, effectively holding the program counter if an undefined input is encountered. This setup allows for quick and efficient resolution of jump addresses, crucial for implementing loops and conditional branches, thus significantly enhancing the processor's control flow management during execution.

Schematic


PC_LUT:pl1
Decoder0
how_high[2..0]
IN[2..0]
OUT[7..0]
2
3
1
WideOr2
2
0
7
6
WideOr1
target[3]~not
4
2
0
WideOr0
target[4]~not
Decoder1
2:1
IN[1..0]
OUT[3..0]
7(0)
target[11..0]

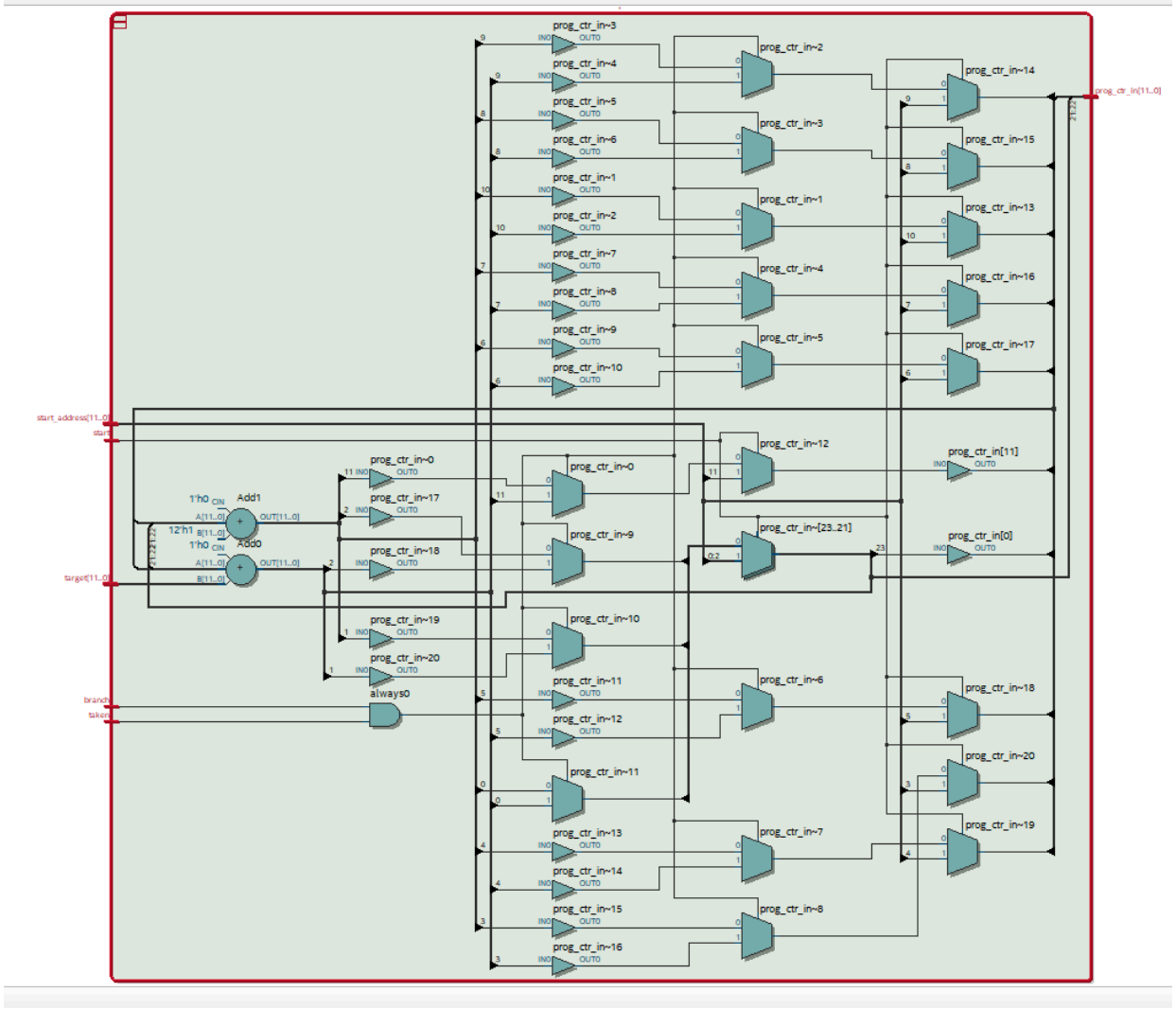
Next PC module

Module file name: nextPC.sv

Functionality Description

The `nextPC` module is designed to dynamically calculate the next value of the program counter within a processor, supporting complex control flows such as conditional branches and initialization. Parameterized by `D=12`, it effectively handles a 12-bit address space, ensuring compatibility with moderately sized instruction sets. The module processes inputs like `start`, `branch`, `taken`, `start_address`, `target`, and `prog_ctr_out` to determine the next program counter value (`prog_ctr_in`). If the `start` signal is active, the program counter is set to the `start_address`, effectively initializing or resetting it. When a conditional branch is indicated (both `branch` and `taken` are true), the program counter jumps to a new location calculated by adding the `target` offset to the current address. In the absence of these conditions, the program counter simply increments by one, moving sequentially to the next instruction. This setup allows the module to manage various jumping and branching mechanisms, thereby enhancing the processor's ability to navigate through different segments of the program based on runtime decisions and initial configurations.

Schematic


start_address[11..0]
start
target[11..0]
branch
taken
1'h0 CIN
Add1
A[11..0]
12'h1 B[11..0]
1'h0 CIN
Add0
A[11..0]
B[11..0]
OUT[11..0]
always0
prog_ctr_in~0
prog_ctr_in~1
prog_ctr_in~2
prog_ctr_in~3
prog_ctr_in~4
prog_ctr_in~5
prog_ctr_in~6
prog_ctr_in~7
prog_ctr_in~8
prog_ctr_in~9
prog_ctr_in~10
prog_ctr_in~11
prog_ctr_in~12
prog_ctr_in~13
prog_ctr_in~14
prog_ctr_in~15
prog_ctr_in~16
prog_ctr_in~17
prog_ctr_in~18
prog_ctr_in~19
prog_ctr_in~20
prog_ctr_in~[23..21]
prog_ctr_in[11]
prog_ctr_in[0]
prog_ctr_in[11..0]
IN0
OUT0

6. Program Implementation

Assembler: code/assembler/assembler.py

Program 1 Pseudocode

```
def hammingDistance(n):

    #x = n1 ^ n2
    #setBits = 0

    #while (x > 0) :
        #setBits += x & 1
        #x >>= 1

    #return setBits
    max = 0
    min = 16
    for i in range(len(n)-1):
        for j in range(i+1, len(n)):
            x = n[i] ^ n[j]
            setBits = 0

            while x > 0:
                setBits += x & 1
                x >>= 1
```

```
            if setBits>max:
                max = setBits
            elif setBits<min:
                min = setBits
    return max, min
if __name__=='__main__':
    n = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    print(hammingDistance(n))
```

Program 1 Assembly Code

code/assember/prog1.txt

Obtaining Data:

- Initialization: The code begins by initializing indices and setting up constants in registers. Values such as min and max are set directly into memory using specific addresses, and the base address for the list n and counters for loop iterations are also established.
- Data Loading: Throughout the loops, the code retrieves elements of the array n by loading them into registers (lw). For example, lw r2 r7 loads the value at the memory address in r7 into r2. This is used to dynamically access array elements based on the current indices i and j controlled by the loop logic.

Manipulating Data:

- Calculating Hamming Distance: For each pair (i, j), the Hamming distance calculation is initiated by performing a bitwise XOR (xor r0 r4) between the elements, which results in a new value where each bit is set if the corresponding bits of the input numbers are different. This result is then manipulated to count the set bits:
    - Counting Set Bits: A series of shifts and bitwise operations (rst, add, xor) iteratively reduce the result while counting bits set to 1. This involves shifting the result right, isolating the least significant bit, and adding it to a counter (setBits).

- Branching and Control: Conditional jumps (jmp, pos) are extensively used to control the flow of the program, especially for managing loop exits and jumping back to loop starts based on conditional evaluations.

Storing Results:

- Updating Maximum and Minimum: After computing the Hamming distance for each pair, the code compares this value against the previously stored max and min values using conditional logic. If conditions are met (e.g., pos r0 000 checks if the result is positive), the new max or min is updated by storing the current setBits back into the respective memory location.
- Memory Writes: Updates to memory (sw) are used to overwrite the max and min values in their respective memory slots whenever a new maximum or minimum Hamming distance is found. This ensures that at the end of execution, the memory locations for max and min hold the correct results.
-

Subroutine and jumps:

- The code uses direct jumps (jmp) and conditional jumps (pos) extensively to manage loop iterations and exit conditions. These are crucial for implementing the nested loop logic and the conditional updates of max and min.

Program 1 Machine Code

code\assembler\mach_code1.txt

Program 2 Pseudocode

```
d# let n be an array of length 64 with 8-bit elements
# do multiply in an order of b2*a2, b2*a1, b1*a2, b1*a1
# a1 is multiplier
def doublePrecisionMult(array):
    for pairOffset in range(16): # 16 pairs of multiplicant and multipliers
        signExtend_subtract = 0b00
        a1 = array[pairOffset + 0]
        a2 = array[pairOffset + 1]
```

```
        b1 = array[pairOffset + 2]
        b2 = array[pairOffset + 3]

        f1,f2 = multiply(b2,a2,signExtend_subtract)
        signExtend_subtract += 1
        g1,g2 = multiply(b2,a1, signExtend_subtract)
        signExtend_subtract += 1
        h1,h2 = multiply(b1, a2, signExtend_subtract)
        signExtend_subtract += 1
        i1,i2 = multiply(b1, a1, signExtend_subtract)
        array[64 + pairOffset + 3] = f2
        array[64 + pairOffset + 2] = f1 + g2 + h2
        array[64 + pairOffset + 1] = h1 + g1 + i2 + overflow
        array[64 + pairOffset + 0] = i1 + Overflow


def multiply(multiplicant, multiplier, signExtend_subtract):
    accumulator = 0
    for i in range(8): # repeat 8 times
        if (multiplier[0] == 1):
            if(signExtend_subtract & 0b01): #if subtract bit is on
                accumulator -= multiplicant
            else:
                accumulator += multiplicant
        multiplier >> 1
        multiplier[7] = multiplicant[0]
        multiplicant >> 1
        if(signExtend_subtract & 0b10): #if signExtend bit is on
            multiplicant[7] = 1
```

```
    return multiplicant, multiplier
```

Program 2 Assembly Code

code/assember/prog2.txt

Obtaining Data:

- Initialization: The code sets up initial values for max and min, placing them in specific memory addresses. Registers are loaded with initial values to start the loops and to prepare constants used in comparisons.
- Loop Setup: The assembly code initializes loop counters and prepares indices for iterating through the array n. These indices (i and j) are stored in memory and are incremented as the loops progress.

Manipulating Data:

- Calculating Differences: For each pair (i, j), the difference is calculated by summing n[i] and the two's complement of n[j] (which is effectively subtracting n[j] from n[i]). The calculation is straightforward: bitwise inversion of n[j] followed by an addition operation.
- Absolute Value Calculation: After computing the difference, the assembly checks if the result is negative (indicating the absolute value needs to be taken). This is done by testing the sign and conditionally negating the result if negative.
- Comparing and Updating Maximum and Minimum: Each calculated difference is then compared with the current max and min values. If the current difference exceeds the max or is less than the min, the respective value is updated in memory.

Storing Results:

- Memory Writes: Updates to max and min are written back to their respective memory locations whenever the new calculated difference requires it. The updates ensure that at the end of execution, the memory holds the correct maximum and minimum differences.

- Loop Controls and Memory Access: The assembly utilizes jump instructions (jmp) for looping control, checking loop conditions, and managing the loop exits. Memory read (lw) and write (sw) instructions are heavily used to load values for processing and to store results or updated loop counters.

Subroutines and Jumps:

- Loop Handling: The assembly code uses conditional and unconditional jumps to handle the nested loops efficiently. It ensures that each element is paired correctly without redundancy, looping back to the start of the inner or outer loop as needed.
- Conditional Execution: Conditional checks are used not only for looping but also for deciding whether to update the max and min values or to take the absolute value of a difference.

Program 2 Machine Code

code\assembler\mach_code2.txt

Program 3 Pseudocode

```
# let n be an array of length 64 with 8-bit elements
# do multiply in an order of b2*a2, b2*a1, b1*a2, b1*a1
# a1 is multiplier
def doublePrecisionMult(array):
    for pairOffset in range(16): # 16 pairs of multiplicant and multipliers
        signExtend_subtract = 0b00
        a1 = array[pairOffset + 0]
        a2 = array[pairOffset + 1]
        b1 = array[pairOffset + 2]
        b2 = array[pairOffset + 3]

        f1,f2 = multiply(b2,a2,signExtend_subtract)
        signExtend_subtract += 1
```

```
        g1,g2 = multiply(b2,a1, signExtend_subtract)
        signExtend_subtract += 1
        h1,h2 = multiply(b1, a2, signExtend_subtract)
        signExtend_subtract += 1
        i1,i2 = multiply(b1, a1, signExtend_subtract)
        array[64 + pairOffset + 3] = f2
        array[64 + pairOffset + 2] = f1 + g2 + h2
        array[64 + pairOffset + 1] = h1 + g1 + i2 + overflow
        array[64 + pairOffset + 0] = i1 + overflow


def multiply(multiplicant, multiplier, signExtend_subtract):
    accumulator = 0
    for i in range(8): # repeat 8 times
        if (multiplier[0] == 1):
            if(signExtend_subtract & 0b01): #if subtract bit is on
                accumulator -= multiplicant
            else:
                accumulator += multiplicant
        multiplier >> 1
        multiplier[7] = multiplicant[0]
        multiplicant >> 1
        if(signExtend_subtract & 0b10): #if signExtend bit is on
            multiplicant[7] = 1
    return multiplicant, multiplier

if __name__=='__main__':
    n = [1, 5, 1, 2, 3]
    print(doublePrecisionMult(n))
```

Program 3 Assembly Code

code/assember/prog3.txt

Obtaining Data:

- Initialization: The assembly code initializes the calculation by setting up memory locations for temporary storage of the high and low bits of results, initial values for maximum and minimum values, and loop counters.
- Loading Values: It uses lw (load word) instructions to fetch the multiplier and multiplicant from predetermined memory addresses, which correspond to the array n values at the indices determined by the loop counters.

Manipulating Data:

- Performing Multiplication: The multiplication for each pair is executed in the order specified (b2*a2, b2*a1, b1*a2, b1*a1). It involves:
    - Initialization of Sign Extension and Subtraction Controls: A control variable signExtend_subtract is managed to handle each multiplication phase correctly.
    - Bitwise Operations and Additions: For each multiplication, the assembly code likely uses shifts (rst for right shift) and adds or subtracts based on the sign and overflow considerations, though the exact implementation details for multiply are abstracted in the high-level description.
    - Storing Partial Results: Intermediate results (f1, f2, g1, g2, h1, h2, i1, i2) are calculated and then combined to form the final double-precision result. This involves adding/subtracting and accounting for carry-bits which might result from each multiplication operation.

Storing Results:

- Updating Results in Memory: After calculating the double-precision results for each multiplication, the assembly code stores the results back into a designated part of the array n (starting from index 64 onward). This is managed through direct memory write (sw) commands that place the high and low parts of each result into successive memory locations.

- Loop Management: The loops in the assembly code use counter increments and conditional jumps (jmp) to iterate over pairs in the array and to handle the nested loop logic required for accessing and updating each array pair's multiplication results.

Subroutines and Jumps:

- Conditional Execution for Multiplication Control: The assembly uses conditional checks and modifications to the signExtend_subtract flag to manage how each multiplication is performed, especially regarding how the results are extended or adjusted based on overflow or negative results.
- Loop Control: The assembly manages loop execution using counters for i and j indices, adjusting them with add and controlled by `jmp` commands based on the array's length and the required iterations for accessing all pairs.

Program 3 Machine Code

code/assember/prog3.sv

Testbenches

For Program 1 and Program 2, which deal with calculating minimum and maximum distances, the test benches follow a systematic approach. They begin by initializing a clock signal that drives the circuits' operations and use a start signal to control the processing of data loaded from external files into the device under test (DUT). These files contain sets of data pairs which the DUT processes to compute distances. The test benches then calculate these distances internally to establish correct answers, comparing these against the DUT's computed results. Discrepancies are logged with detailed error messages, allowing for precise debugging and validation of the DUT’s functionality.

Program 3's test bench focuses on the DUT's capability to perform double precision two's complement multiplication. It follows a similar setup by loading operands and calculating products internally for validation purposes. The DUT then processes the operands to generate products, which are subsequently compared to the internally calculated results. The emphasis here is on ensuring that the multiplication logic within the DUT is accurate and that it handles binary arithmetic correctly across a range of input values.

In all scenarios, the test benches operate through multiple test cycles, processing different sets of data to ensure the DUT’s consistency and reliability under varied conditions. This repetitive testing strategy helps uncover any intermittent issues and ensures that the circuit operates as expected in real-world applications. Each test cycle involves loading data, computing expected results, initiating the DUT's operation, and then verifying the outcomes, with synchronization managed by carefully timed start signals and monitoring of the DUT’s done signals. The results of these tests are critical, providing confidence that the DUT will perform correctly in its intended environment, thereby streamlining the development and optimization of robust digital circuits.

Testbenches outputs

Acknowledgment

We'd like to give a huge thank you to Professor Eldon John for being such an awesome and patient teacher throughout the quarter. His fun and kind approach to teaching made even the toughest topics enjoyable and easier to understand. Your support and encouragement have meant a lot to us, and we're really grateful for all the effort you put into making this class great. Thanks for an amazing quarter, Professor John!