9-bit Processor Design

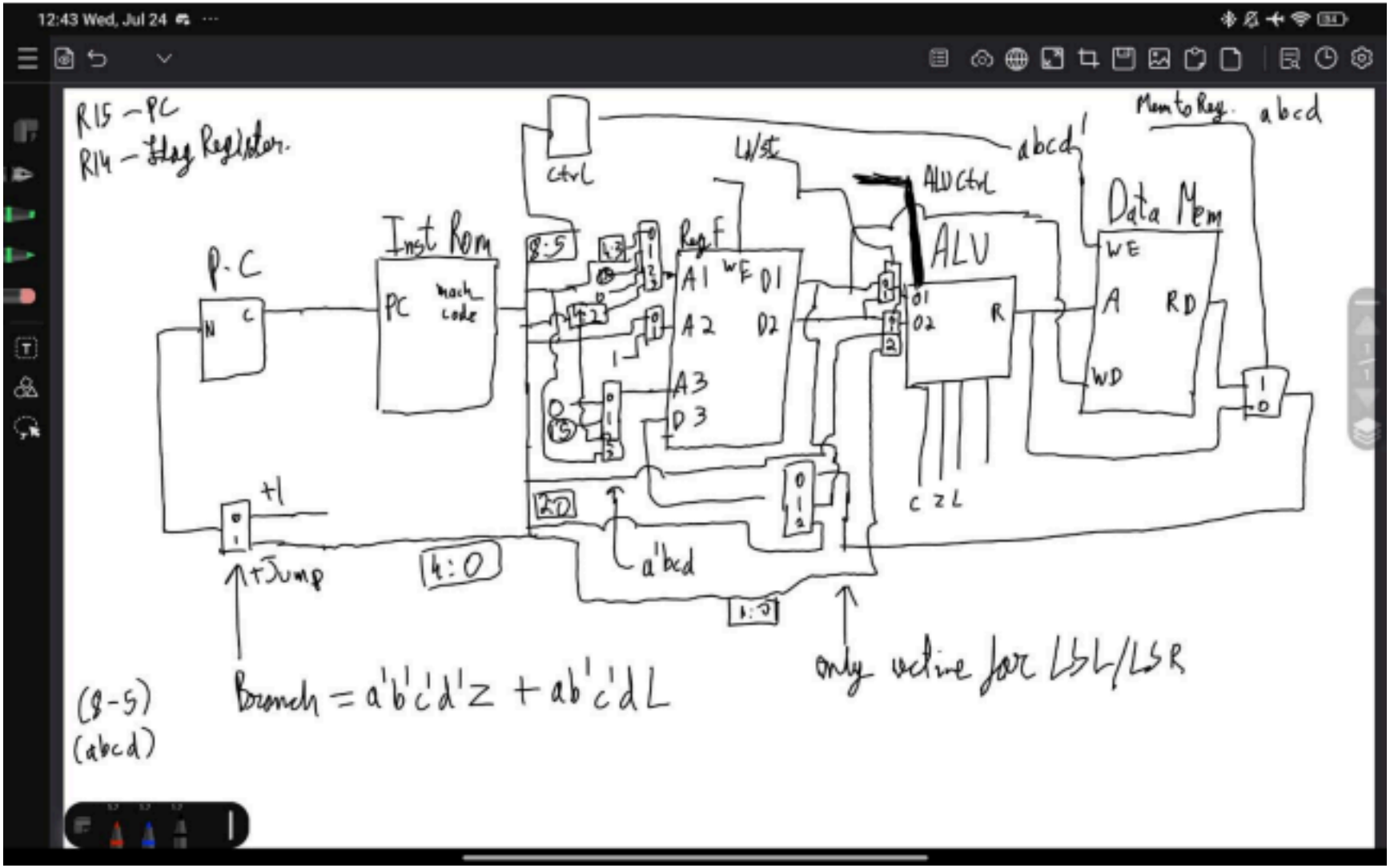
1. Introduction

SORA v1.0

Our overall philosophy for this architecture is to create an architecture that is somewhat similar to ARM, so that it is easy to pick up and master. Most of the instructions we have can be found in ARM as well, with a few additions for our special case. We strived to create an architecture that could handle the 3 programs assigned to us somewhat specifically, designing our ISA completely around the programs, and building on it as we wrote more code. This lead to us having quite a complicated control for our hardware design. Our machine is classified under a standard load-store architecture, where we have a special register used for flags, that we only ever read from for branch and add carry operations. The Flag Register is only ever written to after certain arithmetic, and compare operations.

2. Architectural Overview

We expect to have a Program Counter, Instruction ROM, Pseudo-Control Unit, Register File, Signed Extender for Jumps. Unsigned Extender for Immediates, ALU, Data Memory and muxes to control Data Flow for different operations.



3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	4 bits opcode, 5 bits destination register	add, xor, mov, cmp, cms, adc, rad, not, adf
B	4 bits opcode, 5 bit bit address	blt, beq

I	4 bits opcode, 5 bit register and immediate values	moi, lsl, lsr
M	4 bit opcode, 5 bit address	ldr, str

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
add	R	0001 rr rrr	r2: 00100010 r5: 01001010 add r2 r5 r2: 01101100	The first operand is also the destination register which can only be r0-r3. The add operation also sets the carry flag if there was a carry during the operation.
xor	R	0011 rr rrr	r2: 00100010 r5: 01001010 xor r2 r5 r2: 01101000	The first operand is also the destination register which can only be r0-r3.
lsl	I	0100 rrr ii	r4: 00010101 lsl r4 #3 r4: 10101000	The first operand is also the destination register.
lsr	I	0101 rrr ii	r4: 00010101 lsr r4 #2 r4: 00000101	The first operand is also the destination register. This also sets the AddFlag used in Program 3. In this case the flag would be set high sinc bit 0 is 1.
mov	R	0110 rr rrr	r2: 00100010 r5: xxxxxxxx mov r2 r5	In this case, the 2nd register is the destination register instead of the first. This is because our goal for the move op was to move data from lower to higher registers.

			r5: 00100010	
cmp	R	0111 rr rrr	r2: 11001010 r5: 11001010 cmp r2 r5 the zero flag will be set to 1	The compare operation sets the zero, less than and carry flags. For this example, r15 becomes 00000001.
scm	R	1000 rr rrr	r2: 11001010 r5: 00110000 cmp r2 r5 less than flag will be set to 1.	The signes compare operation also sets the zero, less than and carry flags which are in r15. For this example, r15 becomes 00000010.
adc	R	1011 rr rrr	r2: 00100010 r5: 11001010 r15: 00000100 adc r2 r5 r2: 11101101	The first operand is also the destination register. adc also checks to see if the carry flag has been set or not and adds a 1 to the result accordingly.
rad	R	1101 rr rrr	r2: xxxxxxxx r5: 01001010 rad r2 r5 r2: 00000011	The first register is the destination register.
not	R	0010 rr rrr	r2: xxxxxxxx r5: 01001010 not r2 r5	The first register is the destination register.

			r2: 10110101	

blt	B	1001 iiiii	PC: 000000001011 r15: 00000010 blt 00100 PC: now becomes value stored at JLUT[4]	This check the less than flag and if it is set, changes the PC accordingly.
beq	B	0000 iiiii	PC: 000000001011 r15: 00000010 beq 00100 PC: now becomes address stored at JLUT[4]	This check the zero flag and if it is set, changes the PC accordingly.
moi	I	1010 rr iii	r1: xxxxxxxx moi r1 010 r1: 00000010	The first operand is also the destination register.
ldr	M	1111 iiiii	r0: xxxxxxxx r1: 00000100 Mem[00000110] = 00001111. ldr 00010 r0: 00001111	The destination register for a load operation is always r0 and the base address is always in r1.
str	M	1110 iiiii	r0: 00001111 r1: 00000100 Mem[00000110] = xxxxxxxx.	The source register for a store operation is always r0 and the base address is always in r1.

			ldr 00010 Mem[00000110] = 00001111	

adf	R	1101 rr rrr	r2: 00100010 r5: 11001010 r15: 00001000 adf r2 r5 r2: 11101101	The first operand is also the destination register. adf checks to see if the add flag has been set or not, only then does it add the 2 numbers and stores the result.

Internal Operands

We use 8 general-purpose registers (R0-R7). R0 is used as the source and destination for memory operations. The only special register we use is a FlagRegister as a separate module in TopLevel, used to store zero, carry, less than, and addf flags.

Control Flow (branches)

We use Absolute branching in our architecture, The target address are stored in a Jump lookup table that has values loaded into into whenever ‘Reset’ is high. The addresses correspond to the appropriate line numbers for our assembly/machine code..

Branch if equal (beq): This instruction checks the zero flag in the Flags Register. If the zero flag is set, it branches to desired address.
Branch if less than (blt): This instruction checks the less than flag in the Flags Register. If the less than flag is set, it branches to desired address.

Addressing Modes

We support an indirect addressing mode only, with positive offsets for locations in memory. The addresses are calculated using the ALU. Our base address is always in r1, to which we add a 5-bit unsigned extended offset.
Example: r1: 00000100 ; ldr 00001 ; r0: now has data stored at DataMem[5].

4. Programmer's Model [Lite]

A programmer can approach our machine mostly like one approaches writing ARM Assembly code, since it is quite similar in nature. A programmer should think about where he wants to store the final results of his operations while writing code with our machine, since we only support 4 destination registers for arithmetic operations. The load all necessary values approach works quite well, given that you mov the data into another register before performing another load, since we only support r0 as destination register. For ALU operations, only 2 registers can be used in one instruction, with the 2-bit address register acting as the destination for the result.

No, we cannot simply copy instructions over from MIPS or ARM, since our architecture, only supports 4 registers as destination registers (Except for mov). Our machine code has 2 bits and 3 bits usually reserved for register numbers, and the 2-bit field is almost always the destination. We overcame this problem by adding a frequently used mov operation that allows us to move data in between registers (from lower to higher mostly), and using add operations to move data from high to low registers.

5.Individual Component Specification

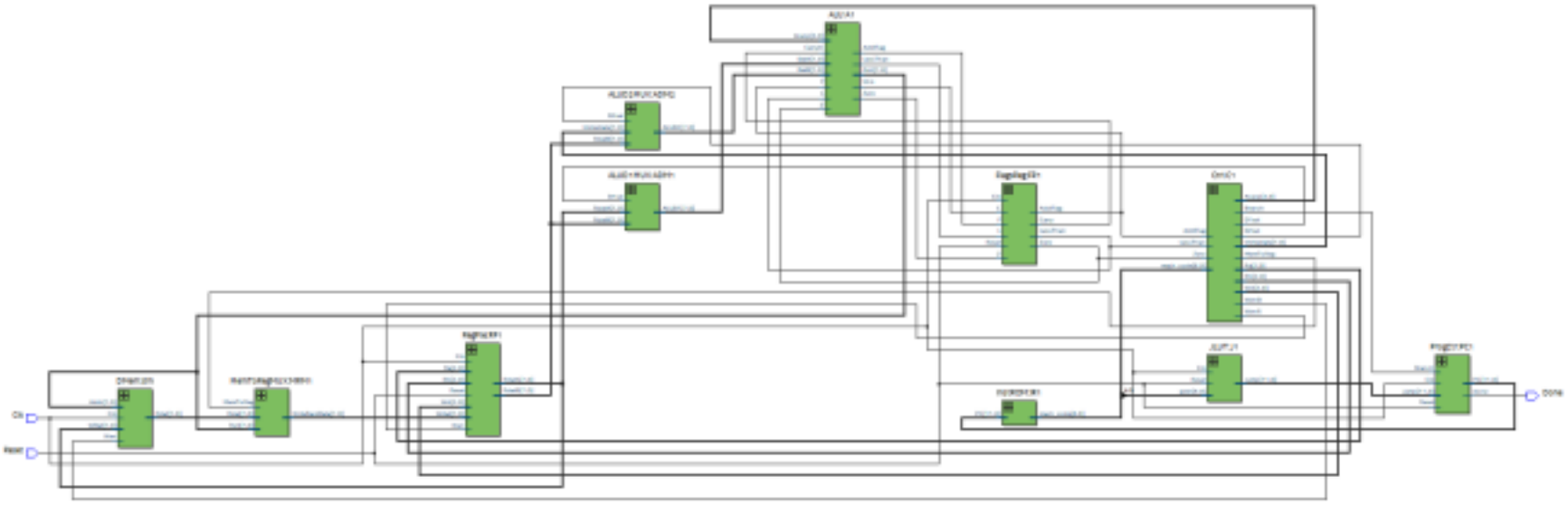
Top Level

Module file name: DUT.sv

Functionality Description

This module brings together all our smaller modules and is the most abstract version of our hardware design. It gives a visual overview of our datapath, controlflow and instruction fetch

Schematic



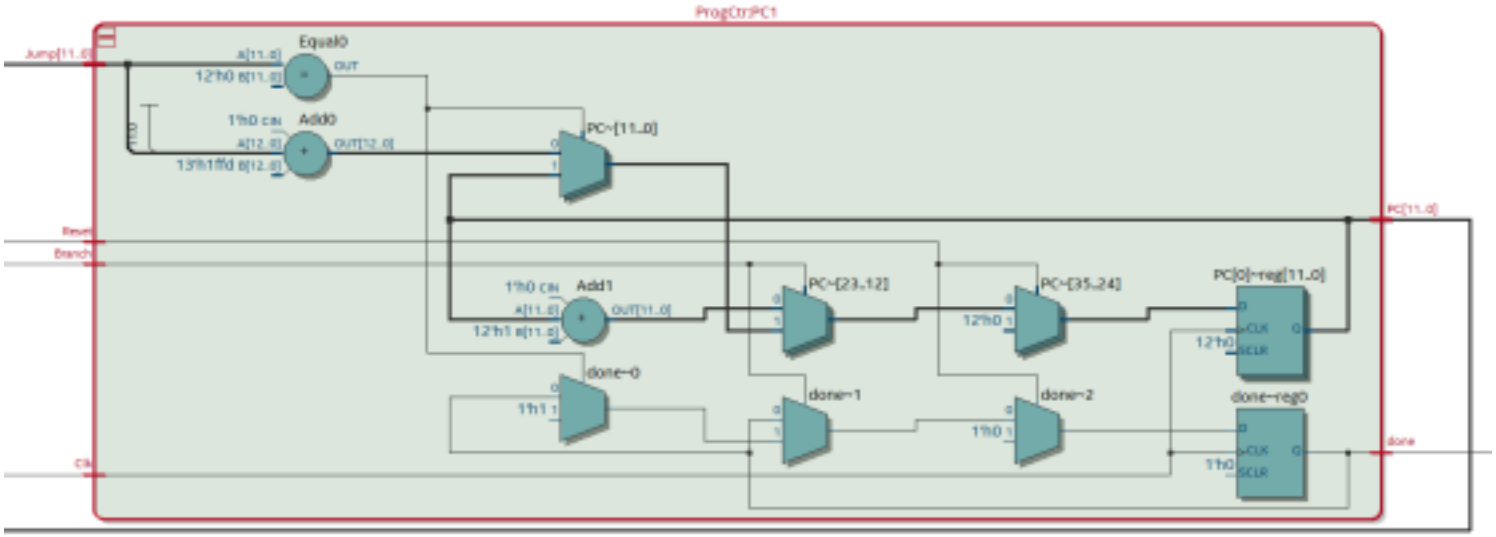
Program Counter

Module file name: ProgCtr.sv

Functionality Description

This module is responsible for outputting the address for the operation we wish to perform in the current cycle. We also handle our branches here, with the control unit setting the branch flag. We usually go to current PC+1 for the next cycle, but for branches, we go to current PC+signed Jump value. The Jump value is sign extended.

Schematic



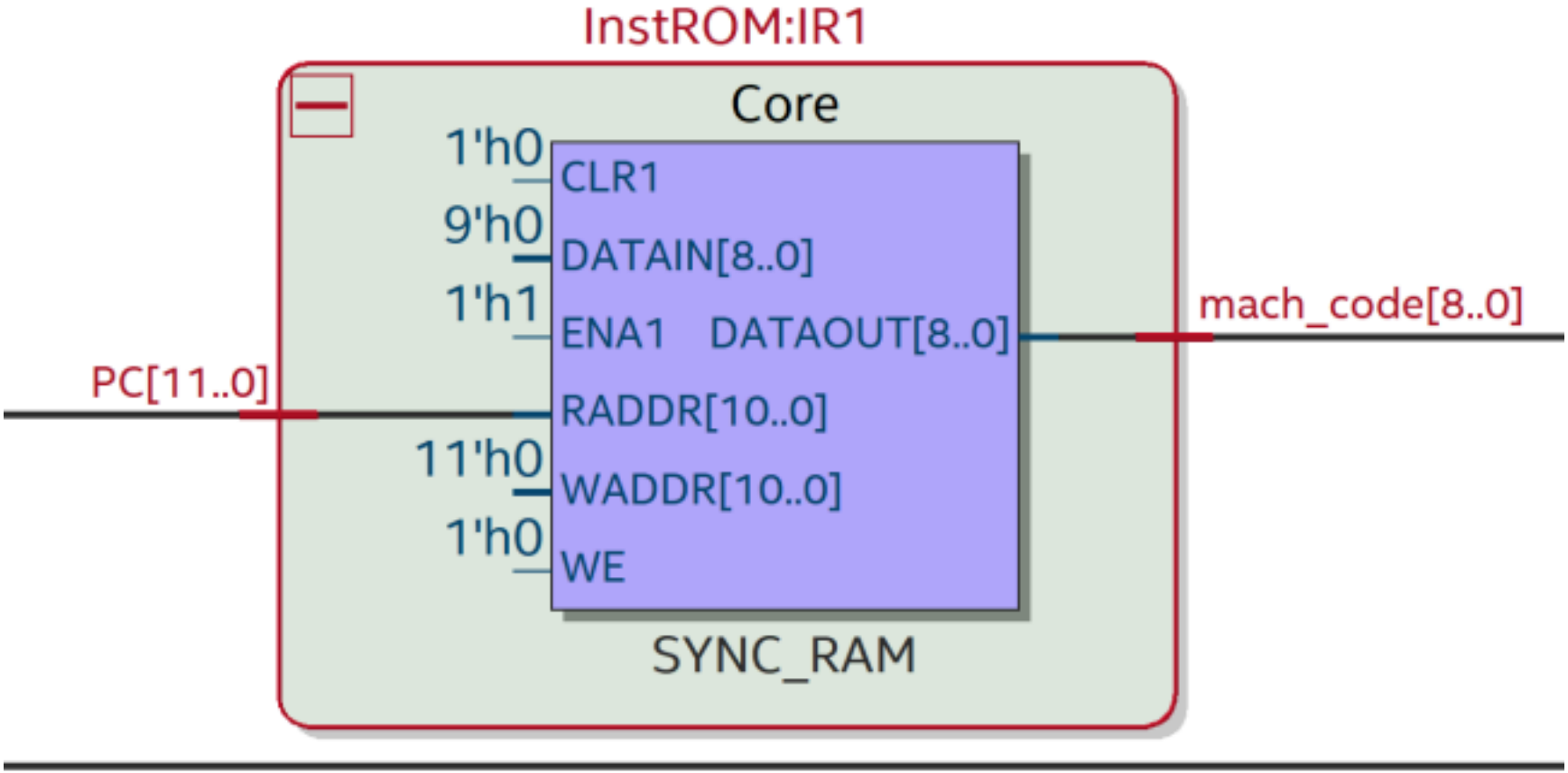
Instruction Memory

Module file name: InstrROM.sv

Functionality Description

This module holds the machine code for our program. It receives the current PC value as input, and outputs the machine code instruction at that address.

Schematic



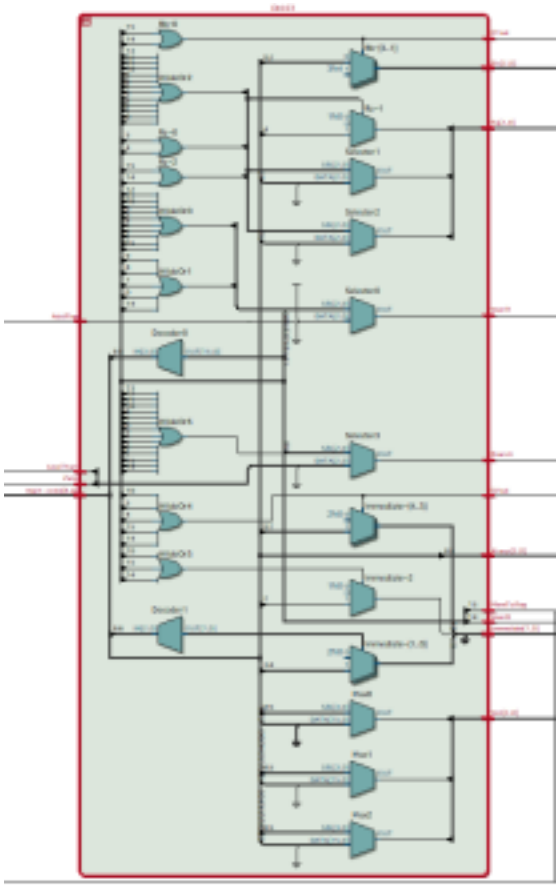
Control Decoder

Module file name: Ctrl.sv

Functionality Description

This module is responsible for setting all Multiplexor selects, Write emnables, Register Addresses and ALU operations. It takes in the machine code (opcode), and certain bits from the Flag Register as input. It then sets the select values, and ALU operation code accordingly.

Schematic


(Sorry for the small text, it wouldn't fit otherwise)

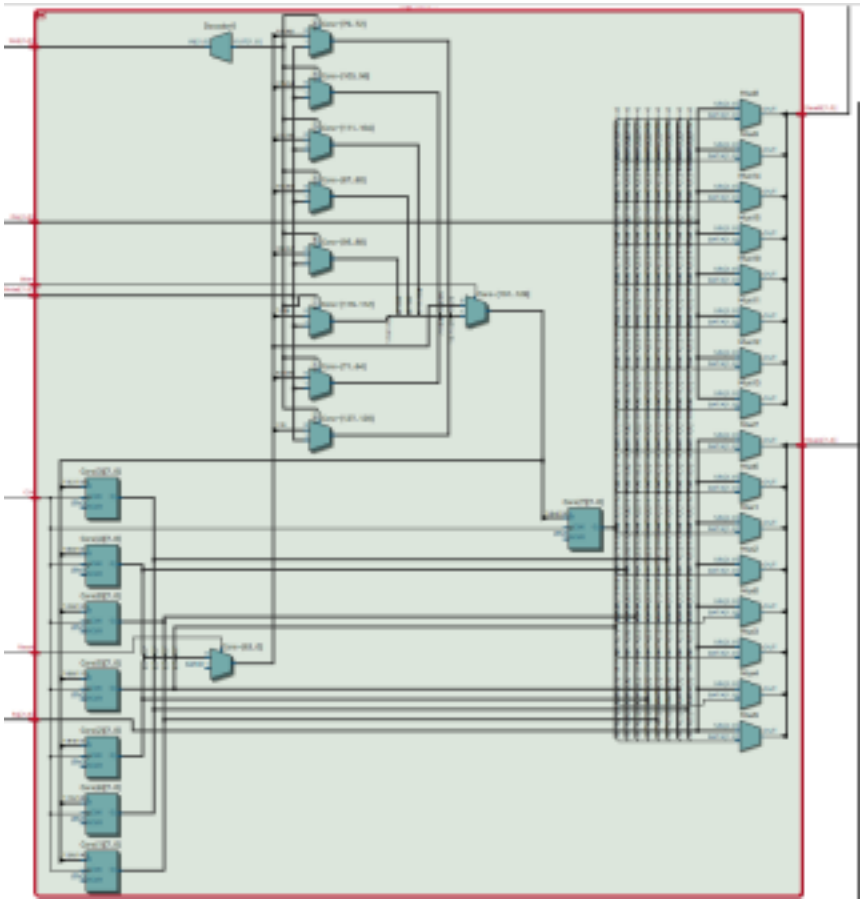
Register File

Module file name: RegFile.sv

Functionality Description

This module is our Register File, It has 8 registers in it. We use an external flag register, only written to and read from for certain operations. It has 3 Address Inputs, 2 Data Outs, 1 Data In, 1 Write Enable.

Schematic



ALU (Arithmetic Logic Unit)

Module file name: ALU.sv

Functionality Description

This module is responsible for performing all Arithmetic and Logical Operations. It takes in 2 Operands as input, and an ALUcode as input which tells the ALU what it needs to do with the 2 operands. It gives only 1 output which is the result.

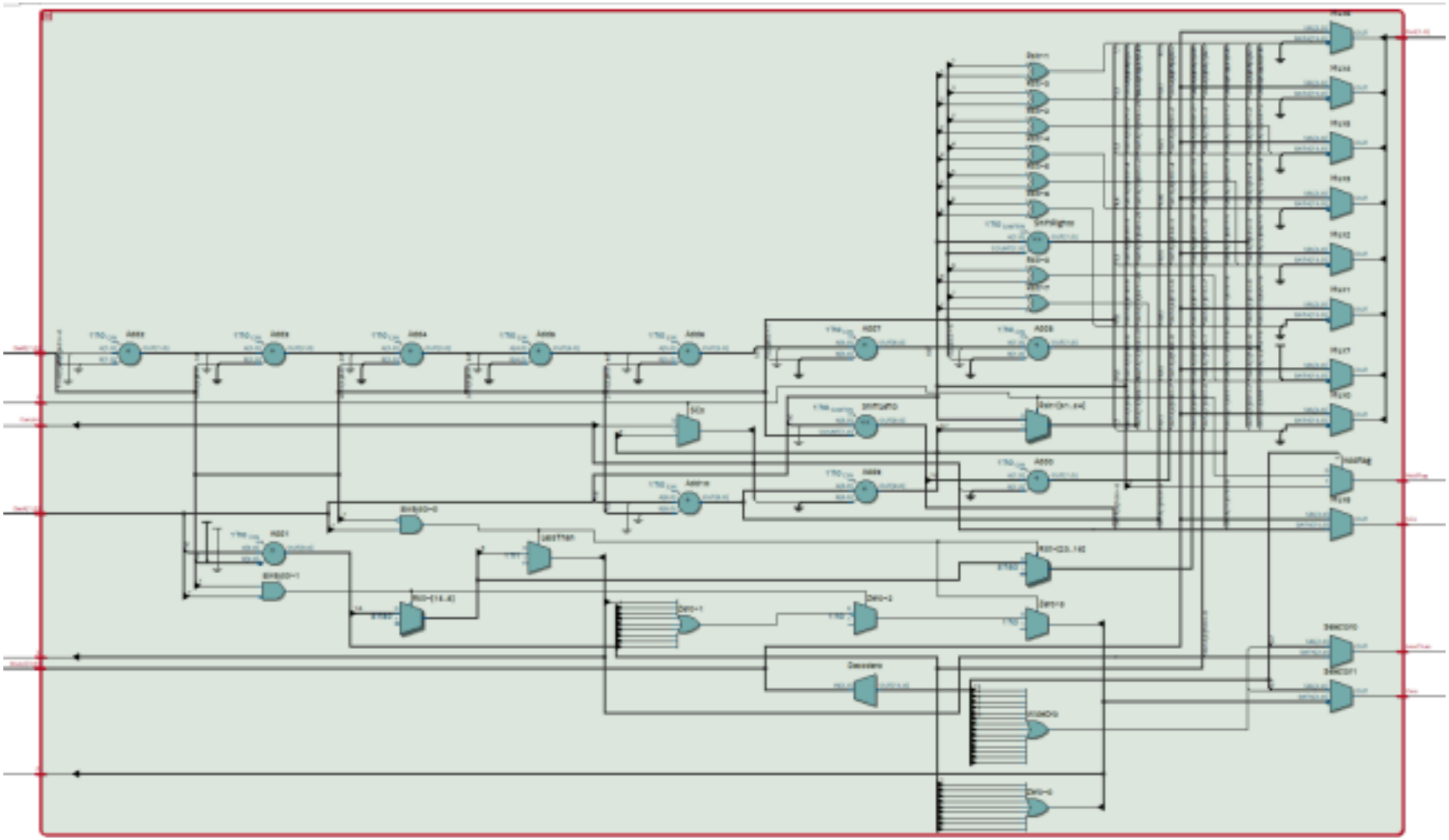
ALU Operations

The Operations are:

- ldr, str, add, adc, adf: addition.
- rad: adds all bits together.
- not: flip all bits.
- xor: bitwise xor.
- lsl, lsr: logical shift left or right.

- cmp, cms: subtraction for setting flags according to result.
- Branches, moves: no ALU involvement.

Schematic



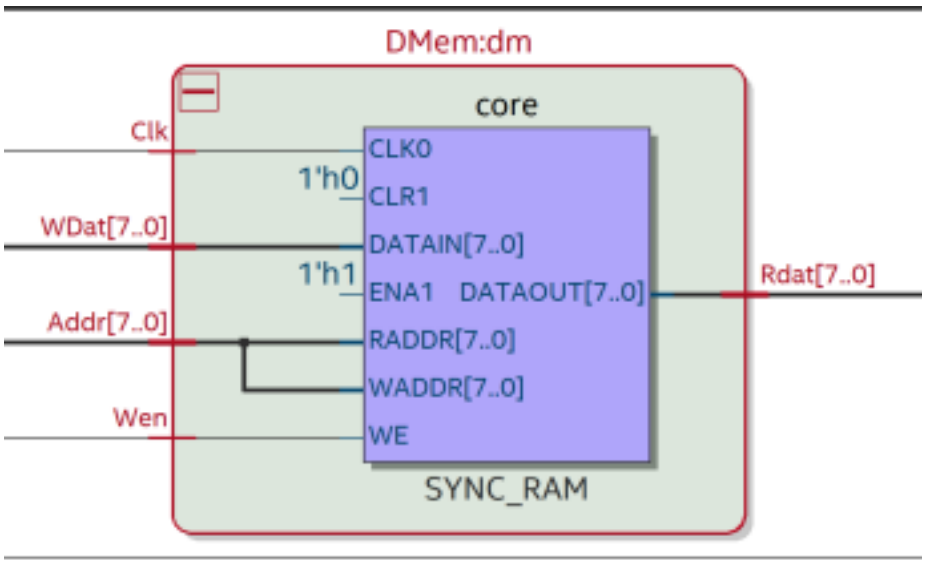
Data Memory

Module file name: DMem.sv

Functionality Description

This module is used as our Memory for the CPU. It has 1 Write Enable Input, 1 Write Data Input, 1 Address Input, 1 Data Output. It stores data needed fro our program, which we fetch using loads. In it we store our final results as well.

Schematic



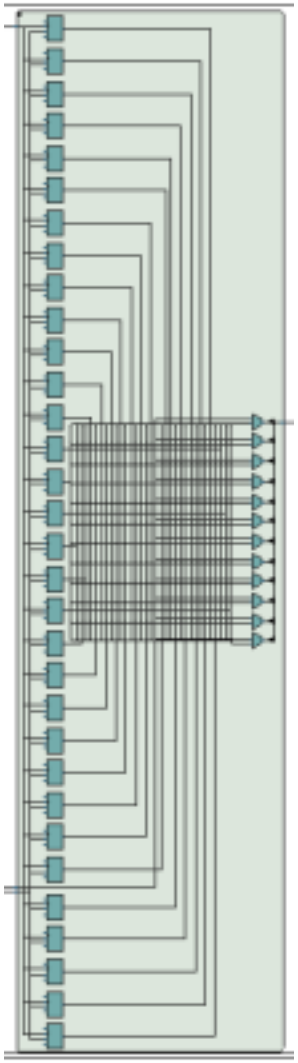
Lookup Tables - Jumps

Module file name: JLUT.sv

Functionality Description

We use this jump look-up table handle control flow(branch) operations. The Look-up table contains 32 addresses, as we use a 5-bit index value. The jump addresses are loaded into the lookup table when ‘Reset’ is set high.

Schematic



Muxes (Multiplexers)

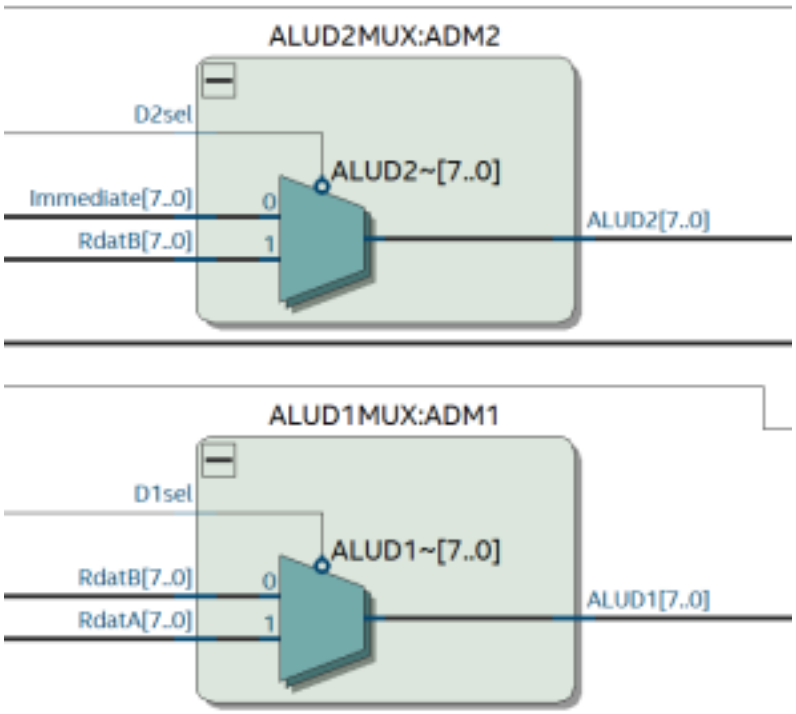
Module file name: ALUD1MUX.sv, ALUD2MUX.sv, MemToRegMUX.sv

Functionality Description

We use 3 Multiplexors in our design:

1. ALUD1MUX: Determines what goes into Data A of the ALU. It chooses between RdatA and RdatB from the RegFIle.
2. ALUD2MUX: Determines what goes into Data B of the ALU. It chooses between RdatB and an Immediate Value. 3. MemToRegMux: Determines what data is written into the Reg File, chooses between the Rslt of the ALU and the data retrieved from DMem.

Schematic



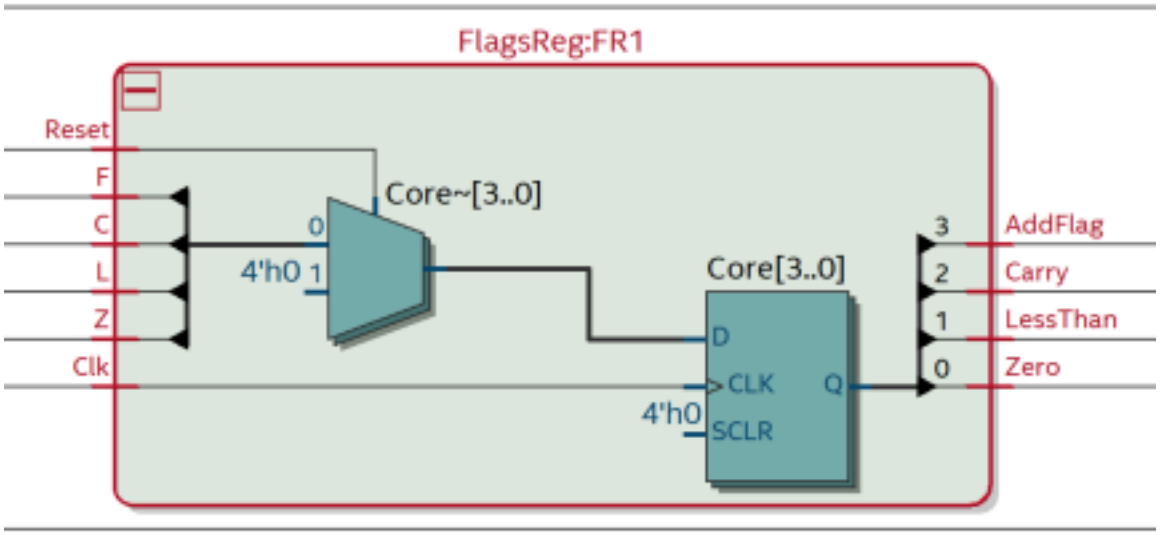
Other Modules (if necessary)

Module file name: FlagsReg.sv

Functionality Description

This is our external Flag Register, we use this to hold Flags needed for adc, branch and adf operations. The way we have deigned it is that it holds the values of the flags for more than one cycle and the values are only ever altered by instructions that are allowed to. Only compare, lsr, lsl, add, adc, adf can alter the flag values, over the other instruction the values are preserved.

Schematic



6. Software

Program 1 Pseudocode

Program 1 Assembly Code

Program1.s

Program 2 Pseudocode

Program 2 Assembly Code

Program2.s

Program 3 Pseudocode

- Program will be executed in 2 halves

- load LSB multiplier - move to R7
- load LSB multiplicand - move to R6
- load MSB multiplicand - move to R5
- R4 will also be for multiplicand
- while R7 ! = 0 - this loop MUST execute 8 times ***
    - If lsb of R7 is 1
    - add 24 bits ({R4, R5, R6}) to ({R1, R2, R3}) (need to handle carrys here) ■ In the last step here we might have 1 carry out which wants to go to R0
    - LSR R7 by 1
    - LSL ({R4, R5, R6}) by 1
- Here, LSB of result (R3) is ready and we can store it by moving it to R0 and str to appropriate memory location.
- We also know now that R6 is all zeroes, so shift them down by 8 (R5→R6, R4→R5, need to set R4 to zeros) ○ Similarly R2→R3, R1→R2, need to set R1 to zeros (or there may be a carry needed here)
- load MSB multiplier into R7
- while R7 ! = 0
    - if lsb of R7 is 1
    - add 24 bits ({R4, R5, R6}) to ({R1, R2, R3})
    - LSR R7 by 1
    - LSL ({R4, R5, R6}) by 1
- The most sign 3 bytes of our answer are now in ({R1, R2, R3})
- move top R0 step by step and store into mem accordingly

Program 3 Assembly Code

Program 3.s

Assembler

We chose to write our assembler in Python, since that seemed to be the easiest language to do so in, given the short duration we have for this project. It contains 4 files: main.py, parse.py, lexer.py and code_generator.py. The final output of the assembler is our machine code to be used for Programs 1, 2 and 3.

Command to Assemble the code: main.py <Assembly file name> <output file name>.

Example: main.py Program1.s mach_code.txt

Program 1 Machine Code

mach_code1v2.txt

Program 2 Machine Code

mach_code2.txt

Program 3 Machine Code

mach_code3v2.txt