Lab 4 Report — Design of Decode, Control FSM, Program Counter, and Associated Datapath Units

Computer Design Laboratory ECE 3710
Fall 2021
The University of Utah

Jacob Peterson
Computer Engineering 2022
University of Utah
Salt Lake City, UT

Brady Hartog
Computer Engineering 2022
University of Utah
Salt Lake City, UT

Isabella Gilman
Computer Engineering 2023
University of Utah
Salt Lake City, UT

Nate Hansen
Computer Engineering 2023
University of Utah
Salt Lake City, UT

***Abstract*—This report focuses on the final design steps of the 16-bit CompactRISC (CR16) processor. The module we designed integrates the datapath, register file, and Block RAM (BRAM) to a Finite State Machine that controls the loading, decoding, and execution of program instructions. We can decode and execute instructions in a single state, and then transition based on the instruction type. To manage the exceptional control flow, we implemented a program counter that has various functions in addition to incrementing after each execute state. We made a few adjustments to our implementation of the datapath, detailed hereafter. We implemented a custom ISA, originally derived from the University of Utah's ECE 3710 class ISA. We will briefly address how we plan to compile assembly instructions from the ISA. We intend to use this project as the processor for a Fully-Synchronized Synthesizer (FSS) prototype.**

I. Introduction

Throughout the design process for the CR16 processor, we have come to understand and adapt its functionality towards our specific application in the FSS. As such, we have modified various aspects of our previous incremental design multiple times. We will report on these changes and decisions, and hopefully provide some insight into our design process. We defined a few capabilities that we need our CR16 processor to have.

1) The CR16 processor must be able to interface with the GPIO banks and the I^2C bus on the FPGA board.
2) The processor must be fast enough to interface with the audio CODEC over I^2C, poll for the state of the button and rotary encoder inputs on the FSS, and update the state of LEDs on the FSS in a reasonable timeframe.
3) The ISA we construct for the CR16 should be something that makes sense to every group member. Instructions should be intuitive, and the FSM implementation of each instruction type should operate efficiently.
4) The conventions and rules of the ISA and the hardware should be well-defined for ease of code development.

Using this short set of goals, we set out to integrate BRAM, the datapath, and the program counter with an FSM that would control the flow of a preloaded software program.

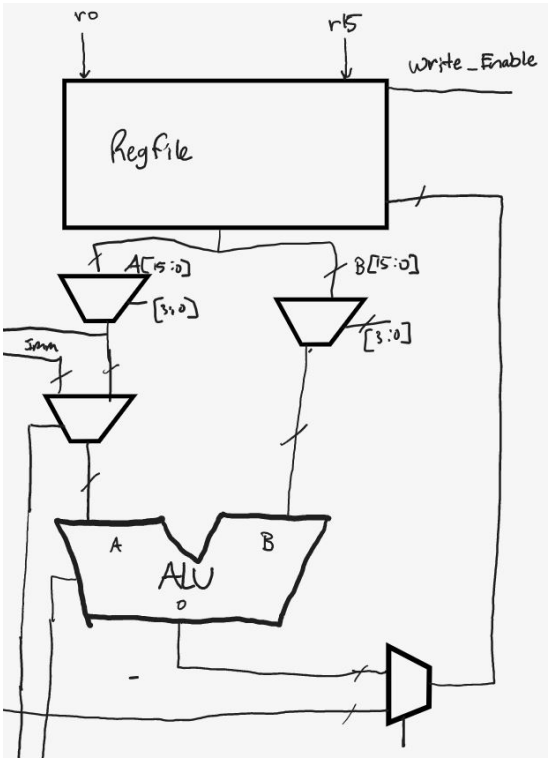

Fig. 1. The most recent block diagram of our CR16 datapath. Some labels removed for simplicity.

II. Changes to the Datapath

We made various changes to our previous implementation of the datapath in order for it to function properly in concert with the FSM. The block diagram of our most recent design for the datapath is contained in Fig. 1. The modified modules were tested with their respective testbenches, which in some cases needed to be modified as well. The changes were as follows:

1) *Added a multiplexer with the ALU output data line to directly load data into the regfile and bypass the ALU.*

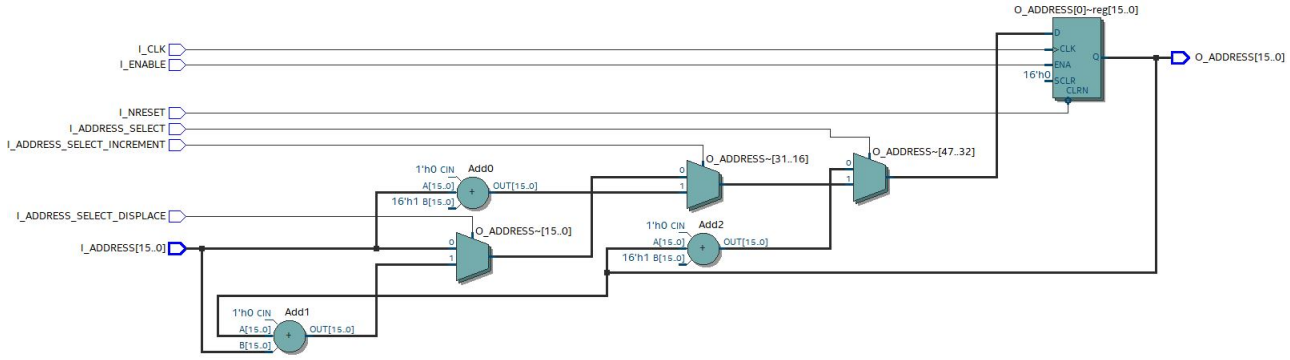

Fig. 2. Block diagram for the program counter module

This was done to facilitate `LOAD/MOV` type instructions.

2) *Consolidate ALU Opcodes.* The way we had initially built our ALU to set flags was flawed for various reasons. We established a new method to do that more efficiently. We eliminated some ALU opcodes that were repetitive, and only the opcodes corresponding to `SUB`, `ADDC`, and `ADD` will set flags. Since writeback to the regfile is a concern of the FSM, the `CMP` and `CMPI` instructions will use the `SUB` opcode on the ALU to trigger the appropriate flags. The three instructions mentioned above will modify all the applicable flags. However, for the sake of consistency in the code, we should only use `CMP/CMPI` by convention for setting `Jcond/Bcond` condition codes.
3) *Moved multiplexing immediate operands to input 'A'.* By convention, we have assigned `Rdest` to correspond to ALU input 'B' and `Rsrc` to correspond to ALU input 'A'. It is a concern of the FSM to select an immediate operand from the instruction or a data value from the regfile to place on ALU input 'A'.
4) *Add a direct output from the regfile at read port 'A'.* This direct output allows the `STORE` instruction to access the regfile directly and select the register data to store in memory through `Rsrc`.

These changes have significantly cleaned up our code and allowed us to set some programming conventions. We hope this will make the programming process as smooth as possible.

III. Design Decisions for the ISA

In order to proceed with the design of the FSM, we need to establish an exact ISA. This will allow us to write logic to decode instructions properly. We made a few modifications to the ISA provided to us so we could implement some desired behavior.

1) `LOADX/STOREX`: These instructions are external `LOAD` and `STORE` instructions which will be capable of storing data directly into an external register. This was implemented for two reasons. We want `LOAD` and `STORE` to have access to the whole address space of instantiated BRAM, and we want to be able to communicate in a modular way with GPIO resgisters that are sending information to the audio CODEC over I^2C.
2) `CALL/CALLD/RET` *instead of* `JAL/JUC`: Our software will potentially be quite long, and those of us who will be writing it want to use the stack to manage subroutine calls within subroutine calls. It seems to make more logical sense to encapsulate the management of the stack within an instruction. The `CALL` and `CALLD` (`CALL` with displacement) instructions will manage the return address implicitly, and `RET` will be an unconditional jump to the previously calculated return address. This way, we don't need a dedicated register for the return address.
3) `PUSH/POP`: These instructions will encapsulate the behavior of modifying the stack pointer and loading/storing data in the stack. Although we haven't implemented them yet, we believe this instructions could be useful.

It is important to note that if the state machine in the FSM becomes too complicated due to the behavior of any of these instructions, we will fall back on the simpler behavior of the provided ISA. The software is likely to be the most burdensome aspect of our project, so we want to develop it intuitively and cleanly. The full ISA is located in Appendix Table 1.

IV. The Program Counter

The program counter module is designed to increment the program counter naturally after the execution of each instruction. This module is depicted in Fig. 2. Multiple complications to the advancement of the program counter arise from the non-linear control flow of jumps, branches, and method calls. In order to handle this, the program counter module has some additional logic. Our program counter supports both direct modification of the program counter and 2's complement signed displacement of the program counter. The input `I_ADDRESS_SELECT` multiplexes between the natural incrementing of the PC and a user-defined input as seen on `I_ADDRESS`. The `I_ADDRESS_SELECT_DISPLACE` multiplexes between the natural incrementing of the PC and adding a 2's compliment displacement seen on `I_ADDRESS`.

The input I_SELECT_INCREMENT multiplexes between propagating I_ADDRESS to O_ADDRESS and propagating I_ADDRESS + 1 to O_ADDRESS. We implemented the increment selector as a way to return to the correct address after a method call. This may be done in other ways, but this seemed to simplify the FSM code. This PC setup should allow for all types of conditional control flow available in our ISA.

V. Decoding and Executing Instructions

With all of the opcodes and instruction behavior established, we are prepared to design and implement an FSM that will fetch, decode, and execute instructions that are pre-loaded to the BRAM module through a memory initialization file.

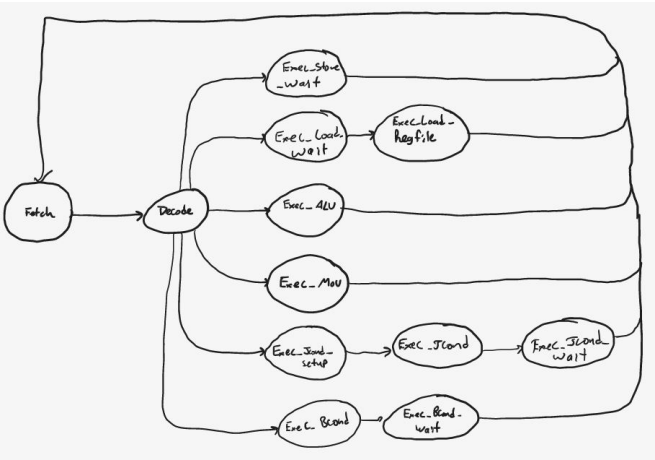

Fig. 3. State Transition Graph (STG) of the FSM

The finite state machine follows the State Transition Graph as depicted in Fig. 3. The instructions are decoded combinationally on the fetch state. The three types of instructions take varying numbers of clock cycles from fetch to execute. ALU instructions, both R-type and I-type, take three clock cycles to execute. The FSM sets the selector that distinguishes register-to-register instructions from immediate instructions. It also allows the output of the ALU to pass to the write port. On the final execute state, the write port is enabled for Rdest. The connections for these various forms of data transfer can be seen in Fig. 4.

The STORE and MOV instructions also take three clock cycles to execute. For STORE, the FSM simply selects Rsrc on the datapath and multiplexes the data at Rsrc to the BRAM write port. For MOV, MOVIL, and MOVIU, the FSM determines whether to write an immediate or data from Rsrc into Rdest. Since instruction immediates are only 8 bits, our instruction decode logic uses two instantiated 16-bit wires to interpret the immediate as both the upper 8 bits and the lower 8 bits. Simple if-then-else logic determines how to move the data.

The LOAD instruction takes four clock cycles. The instruction decoding is combinational, but interfacing with the memory requires some propagation waits. The FSM tells BRAM what address we wish to read from, and then that

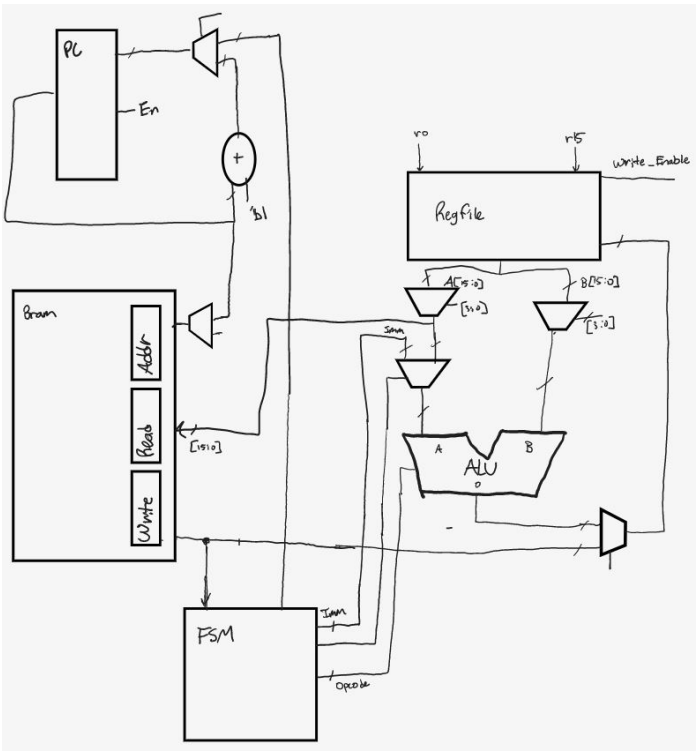

Fig. 4. Full block diagram for the CR16 processor.

data propagates to the multiplexer at the output of the ALU which is set to write data from memory to the regfile.

The Jcond instructions take four clock cycles and Bcond instructions take three clock cycles. The FSM checks to see if the condition is met, loads the target address or displacement onto the input of the PC, and then enables the PC and I_ADDRESS_ENABLE to modify the program counter directly. Jcond takes one step longer than Bcond because of the latency of loading the absolute address out of the regfile and onto the PC module. If the condition is not met, the FSM returns to the fetch stage and the program counter is enabled.

States for CALL/CALLD/RET/PUSH/POP will be implemented in the coming week.

VI. Testing With Software

We wrote a series of test programs designed to test each type of instruction incrementally. We broke those programs down as follows:

1) *ALU instructions:* We wrote a couple of simple programs to test ALU operations.
   a) add_fibonacci.dat: Uses ADD and ADDI instructions to calculate 5 digits of the Fibonacci series.
   b) mul_powers2.dat: Uses MUL and MULI instructions to display powers of 2.
   c) logic.dat: Uses a combination of AND, OR, NOT, XOR, ARSH, ALSH, RSH, and LSH instructions to test logic exhaustively.

Analysis & Synthesis Resource Usage Summary	
Resource	Usage
Estimate of Logic utilization (ALMs needed)	534
Combinational ALUT usage for logic	740
-- 7 input functions	6
-- 6 input functions	291
-- 5 input functions	151
-- 4 input functions	134
-- <=3 input functions	158
Dedicated logic registers	394
I/O pins	59
Total MLAB memory bits	0
Total block memory bits	32768
Total DSP Blocks	1
Maximum fan-out node	I_NRESET~input
Maximum fan-out	413
Total fan-out	5894
Average fan-out	4.59

Fig. 5. Resource Summary from the Quartus mapping synthesis report.

	Input Port	Output Port	RR	RF	FR	FF
1	I_NRESET	O_MEM_ADDRESS[0]	12.539	13.812	12.970	14.277
2	I_NRESET	O_MEM_A...ESS[10]	12.720	13.664	13.114	14.064
3	I_NRESET	O_MEM_A...ESS[13]	12.245	13.035	12.703	13.485
4	I_NRESET	O_MEM_ADDRESS[3]	12.071	12.940	12.569	13.444
5	I_NRESET	O_EXT_ME...DRESS[7]	11.163			12.700
6	I_NRESET	O_MEM_ADDRESS[7]	11.365	12.038	11.823	12.488
7	I_NRESET	O_EXT_ME...DRESS[3]	11.199			12.427
8	I_NRESET	O_EXT_ME...DRESS[0]	11.130			12.404
9	I_NRESET	O_EXT_ME...RESS[11]	11.097			12.313
10	I_NRESET	O_MEM_DATA[15]	10.998			12.296
11	I_NRESET	O_EXT_M...ATA[11]	10.992			12.136
12	I_NRESET	O_MEM_ADDRESS[8]	10.965	11.735	11.303	12.108
13	I_NRESET	O_EXT_ME...RESS[14]	10.817			11.994
14	I_NRESET	O_MEM_A...ESS[12]	10.901	11.581	11.326	11.956
15	I_NRESET	O_MEM_ADDRESS[5]	10.987	11.484	11.407	11.910
16	I_NRESET	O_MEM_DATA[0]	10.548			11.831
17	I_NRESET	O_MEM_ADDRESS[4]	10.846	11.352	11.304	11.802
18	I_NRESET	O_MEM_DATA[11]	10.475			11.659
19	I_NRESET	O_EXT_ME...DRESS[5]	10.672			11.656
20	I_NRESET	O_MEM_ADDRESS[9]	10.845	11.184	11.262	11.607

Fig. 6. Quartus Timing Analyzer output for propagation delay on combinational paths within CR16.

These tests can be viewed in the demo video submitted for the first checkoff.

2) *Memory interfacing instructions:* We wrote two programs to test `MOV`, `LOAD`, and `STORE` instructions. These programs calculate 4 digits of the Fibonacci series and shuffle them between memory and the regfile.
   While testing these instructions we determined that the memory space would use a dedicated stack pointer, defined as `r15` in Table 2 of the appendix. The stack will grow from the highest address in BRAM to the lowest. We should have plenty of memory space so as not do overwrite instructions, but since we use 16-bit addresses, we can always expand our BRAM capacity.
3) *Control flow instructions:* We wrote a bank of 7 tests for each `Bcond` and `Jcond` to exhaustively test each condition. Each test performs some meaningless operations with "for" loops, "while" loops, and if-then-else statements. We have found that each of these works properly.

All tests can be found within the "src/asm" directory of our project repository.

VII. Synthesis Reports

The resource use of the CR16 design is detailed in Fig. 5. Of course we don't occupy nearly the maximum ALUT capacity of the FPGA. We can see clearly that our unit uses 32,768 bits (4KB). This will likely be enough memory to accomplish our needs, but we may not know until we have written our code. Since we have established 16-bit addresses, we can expand the memory if needed.

There is a very large amount of fan-out coming from the master reset signal `I_NRESET`, which is tied to essentially every register in the hardware. There is potentially a lot of power loss that can be attributed to this, but this signal will inevitably have a lot of fan-out. Despite that large fan-out, the average fan-out of the whole system is 4.59. We are not too concerned with the power usage of this device as our attention shifts to the construction of the FSS hardware.

VIII. Timing Analysis

Meaningful timing analysis is a little challenging with sequential logic, so we have run the the timing analyzer on just the CR16 module. We set it to measure just propagation delay, and the results were as expected. Since the fanout is so high on the `I_NRESET` signal, we would expect it to have the longest propagation delays. This can be seen in Fig. 6. Hopefully if we run into any clock and timing issues with our FSM and the 50 MHz FPGA clock, we can get some debugging help, but our current tests have not exposed any potential timing issues in that regard.

IX. Note on the Assembler

Since we heavily modified the provided ISA, we believe it is necessary to write our own assembler. This is not currently complete, so we have no results to report, but we will address briefly some design obstacles that the assembler will have to handle. First, the assembler will have to keep track of line numbers to count displacements properly, so we have to decide how to handle labels in our assembly code. We will also have to decide all the conventions of the assembly language such as how to write registers, whether or not to include commas, and whether or not to lead immediates with a character like a '$'. These decisions will come this weekend as we put together the finishing touches on the CR16. Some of the software convention decisions such as register naming conventions and jump condition encoding are contained in Appendix Table 2 & 3.

X. Conclusion

Our CR16 processor is nearly completely equipped for our application. Moving forward, we have to finish the implementation of `CALL`, `CALLD`, `RET`, `PUSH`,`\verb` and `POP` instructions. These things will be implemented this weekend, along with support for all instructions in our custom assembler. We will also be setting forward a timeline for constructing our device and implementing our software system by the December 9th deadline. The labor in the future will be divided in the following ways:

1) Nate Hansen: Lab 4 report, LUT sine wave software, audio CODEC, test assembly for CR16.
2) Jacob Peterson: CR16 SystemVerilog file, I^2C protocol software, assembly compiler, Github setup and directory control, Bill of Materials (BOM), parts purchase and inventory, PCB design, coordinate demonstration video.
3) Isabella Gilman: SolidWorks drawings, physical design of FSS device, PCB design, device assembly.
4) Brady Hartog: Conceptual design, drawings and dimensions, Parts purchase and inventory, PCB design, materials acquisition, contact consultants.

We are excited to see our CR16 at work with our Fully-Synchronized Synthesizer prototype.

Appendix

CompactRISC16 (CR16) Instruction Set Architecture (ISA)

Table 1: Assembly Instructions and Machine Encodings

					ImmHi/	ImmLo/	
			Opcode	Rdest	Opcode Ext	Rsrc	
Mnemonic	**Operands**	**Function**	**[15:12]**	**[11:8]**	**[7:4]**	**[3:0]**	**Notes**
ADD	Rdest, Rsrc	`Rdest = Rdest + Rsrc`	0000	Rdest	0000	Rsrc	
ADDI	Rdest, Imm	`Rdest = Rdest + Imm`	0001	Rdest	ImmHi	ImmLo	Sign extended Imm
ADDC	Rdest, Rsrc	`Rdest = Rdest + Rsrc + 1`	0000	Rdest	0001	Rsrc	
ADDCI	Rdest, Imm	`Rdest = Rdest + Imm + 1`	0010	Rdest	ImmHi	ImmLo	Sign extended Imm
MUL	Rdest, Rsrc	`Rdest = Rdest * Rsrc`	0000	Rdest	0010	Rsrc	
MULI	Rdest, Imm	`Rdest = Rdest * Imm`	0011	Rdest	ImmHi	ImmLo	Sign extended Imm
SUB	Rdest, Rsrc	`Rdest = Rdest - Rsrc`	0000	Rdest	0011	Rsrc	
SUBI	Rdest, Imm	`Rdest = Rdest - Imm`	0100	Rdest	ImmHi	ImmLo	Sign extended Imm
CMP	Rdest, Rsrc	`Rdest - Rsrc`	0000	Rdest	0100	Rsrc	
CMPI	Rdest, Imm	`Rdest - Imm`	0101	Rdest	ImmHi	ImmLo	Sign extended Imm
NOT	Rdest, Rsrc	`Rdest = !Rsrc`	0000	Rdest	0101	Rsrc	
NOTI	Rdest, Imm	`Rdest = !Imm`	0110	Rdest	ImmHi	ImmLo	Zero extended Imm
AND	Rdest, Rsrc	`Rdest = Rdest & Rsrc`	0000	Rdest	0110	Rsrc	
ANDI	Rdest, Imm	`Rdest = Rdest & Imm`	0111	Rdest	ImmHi	ImmLo	Zero extended Imm
OR	Rdest, Rsrc	`Rdest = Rdest | Rsrc`	0000	Rdest	0111	Rsrc	NOP instruction is OR R0, R0
ORI	Rdest, Imm	`Rdest = Rdest | Imm`	1000	Rdest	ImmHi	ImmLo	Zero extended Imm
XOR	Rdest, Rsrc	`Rdest = Rdest ^ Rsrc`	0000	Rdest	1000	Rsrc	
XORI	Rdest, Imm	`Rdest = Rdest ^ Imm`	1001	Rdest	ImmHi	ImmLo	Zero extended Imm
LSH	Rdest, Ramount	`Rdest = Rdest << Ramount`	0000	Rdest	1001	Ramount	$0 \leq$ Ramount ≤ 15 since registers are only 16-bits
LSHI	Rdest, ImmLo	`Rdest = Rdest << Imm`	0000	Rdest	1010	ImmLo	$0 \leq$ ImmLo ≤ 15
RSH	Rdest, Ramount	`Rdest = Rdest >> Ramount`	0000	Rdest	1011	Ramount	$0 \leq$ Ramount ≤ 15
RSHI	Rdest, ImmLo	`Rdest = Rdest >> Imm`	0000	Rdest	1100	ImmLo	$0 \leq$ ImmLo ≤ 15
ALSH	Rdest, Ramount	`Rdest = Rdest <<< Ramount`	0000	Rdest	1101	Ramount	$0 \leq$ Ramount ≤ 15
ALSHI	Rdest, ImmLo	`Rdest = Rdest <<< Imm`	0000	Rdest	1110	ImmLo	$0 \leq$ ImmLo ≤ 15
ARSH	Rdest, Ramount	`Rdest = Rdest >>> Ramount`	0000	Rdest	1111	Ramount	$0 \leq$ Ramount ≤ 15
ARSHI	Rdest, Imm	`Rdest = Rdest >>> Imm`	1111	Rdest	0000	ImmLo	$0 \leq$ ImmLo ≤ 15
MOV	Rdest, Rsrc	`Rdest = Rsrc`	1111	Rdest	0001	Rsrc	Copies Rsrc into Rdest
MOVIL	Rdest, Lower Imm	`Rdest[7:0] = Imm`	1010	Rdest	ImmHi	ImmLo	Zero extended Imm, moves immediate value into lower bits of Rdest
MOVIU	Rdest, Upper Imm	`Rdest[15:8] = Imm`	1011	Rdest	ImmHi	ImmLo	Zero padded Imm, moves immediate value into upper bits of Rdest
J[condition]	Rtarget	`if [condition]:` `PC = Rtarget`	1111	condition	0010	Rtarget	[condition] bit patterns are in Table 2.
B[condition]	Displacement Imm	`if [condition]:` `PC += Imm`	1100	condition	ImmHi	ImmLo	[condition] bit patterns are in Table 2. Immediate is sign extended 2's complement for program counter/address displacement.
CALL	Rtarget	`Pushes PC onto stack,` `PC = Rtarget`	1111	xxxx	0011	Rtarget	Used for nested subroutines

CALLD	Displacement Imm	`Pushes PC onto stack, PC += Imm`	1101	ImmHi	ImmMid	ImmLo	Used for nested subroutines. Immediate is sign extended 2's complement for program counter/address displacement.
RET		`Pops top of stack into PC, PC++`	1111	xxxx	0100	xxxx	Used to return from nested subroutine
LPC	Rdest	`Rdest = PC + 1`	1111	Rdest	0101	xxxx	Sets Rdest to the current instruction address/PC + 1
LSF	Rdest	`Rdest = status flags`	1111	Rdest	0110	xxxx	Sets Rdest to the current status flags (zero extended)
SSF	Rsrc	`Status flags = Rsrc[4:0]`	1111	xxxx	0111	Rsrc	Sets the current status flags to `Rsrc[4:0]`
PUSH	Rsrc	`rsp--, Main memory value at rsp = Rsrc`	1111	xxxx	1000	Rsrc	Pushes Rsrc onto top of stack
POP	Rdest	`Rdest = Main memory value at rsp, rsp++`	1111	Rdest	1001	xxxx	Pops top of stack into Rdest
LOAD	Rdest, Raddr	`Rdest = Main memory value at Raddr`	1111	Rdest	1010	Raddr	Used to load data at Raddr into Rdest from main memory
STORE	Raddr, Rsrc	`Main memory value at Raddr = Rsrc`	1111	Raddr	1011	Rsrc	Used to store data at Raddr from Rsrc to main memory
LOADX	Rdest, Raddr	`Rdest = External memory at Raddr`	1111	Rdest	1100	Raddr	Used to load data at Raddr into Rdest from external/peripheral memory/registers
STOREX	Raddr, Rsrc	`External memory value at Raddr = Rsrc`	1111	Raddr	1101	Rsrc	Used to store data at Raddr from Rsrc to external/peripheral memory/registers
NOP		`No Operation`					Pseudo instruction for: OR R0, R0

Table 2: Bit Patterns of Conditions for B[condition] and J[condition]

Mnemonic	Bit Pattern	Description	Function	Status Flags
EQ	0000	Equal	`Rsrc == Rdest`	`Z=1`
NE	0001	Not Equal	`Rsrc != Rdest`	`Z=0`
CS	0010	Carry Set	`C == 1`	`C=1`
CC	0011	Carry Clear	`C == 0`	`C=0`
FS	0100	Flag Set	`F == 1`	`F=1`
FC	0101	Flag Clear	`F == 0`	`F=0`
LT	0110	Less Than	`signed: Rdest < Rsrc`	`N=0 and Z=0`
LE	0111	Less than or Equal	`signed: Rdest <= Rsrc`	`N=0`
LO	1000	Lower than	`unsigned: Rdest < Rsrc`	`L=0 and Z=0`
LS	1001	Lower than or Same as	`unsigned: Rdest <= Rsrc`	`L=0`
GT	1010	Greater Than	`signed: Rdest > Rsrc`	`N=1`
GE	1011	Greater than or Equal	`signed: Rdest >= Rsrc`	`N=1 or Z=1`
HI	1100	Higher than	`unsigned: Rdest > Rsrc`	`L=1`
HS	1101	Higher than or Same as	`unsigned: Rdest >= Rsrc`	`L=1 or Z=1`
UC	1110	Unconditional		`N/A`
	1111	Never Jump		`N/A`

Table 3: Register Naming and Conventions

Register Index	Register Name	Meaning
`4'd15`	`rsp`	Stack pointer with an address starting at `0xFFFF` (2^{16}) and grows downward towards dynamically allocated memory
`4'd14`	`r14`	4th subroutine argument
`4'd13`	`r13`	3rd subroutine argument
`4'd12`	`r12`	2nd subroutine argument
`4'd11`	`r11`	1st subroutine argument
`4'd10`	`r10`	Return value of subroutine
`4'd9`	`r9`	Caller-owned
`4'd8`	`r8`	Caller-owned
`4'd7`	`r7`	Caller-owned
`4'd6`	`r6`	Caller-owned
`4'd5`	`r5`	Callee-owned
`4'd4`	`r4`	Callee-owned
`4'd3`	`r3`	Callee-owned
`4'd2`	`r2`	Callee-owned
`4'd1`	`r1`	Callee-owned
`4'd0`	`r0`	Callee-owned