Project 3 Report

Ezekiel Elin

December 8, 2017

ISA Design

Instructions

All instructions take up 32 bits. The instructions are split up into components based on their format.

- Opcode: 7 bits
- Registers: 5 bits each
- Immediate: 15 bits each
- Pointer Immediate: 20 bits each

A-format

31 – 25	24 – 20	19 – 15	14 – 10	09 – 00
Opcode	Dest Register	LHS Register	RHS Register	Unused

B-format

31 – 25	24 – 00
Opcode	Unused

C-format

31 – 25	24 – 20	19 – 15	14 – 00
Opcode	Destination Register	Source Register	Immediate

D-format

31 – 25	24 – 20	19 – 00
Opcode	Register	Pointer Immediate

Instruction List

Instruction Name	Format	Opcode (decimal)	Pseudocode Eq.	Notes
NOP	B	0	N/A	
ADD	A	1	R1 = R2 + R3	
	B ← R[IR<14..10>] C ← R[IR<19..15>] + B R[IR<24..20>] ← C			
SUB	A	2	R2 = R2 − R3	
	B ← R[IR<14..10>] C ← R[IR<19..15>] − B R[IR<24..20>] ← C			
ADDI	C	3	R0 = R2 + 8	
	B ← IR<14..00> C ← R[IR<19..15>] + B R[IR<24..20>] ← C			
SUBI	C	4	R0 = R1 − 16	
	B ← IR<14..00> C ← R[IR<19..15>] − B R[IR<24..20>] ← C			
ADDS	A	5	R1 = R2 + R3	Sets the flags based on the result
	B ← R[IR<14..10>] C ← R[IR<19..15>] + B R[IR<24..20>] ← C			
SUBS	A	6	R2 = R2 − R3	
	B ← R[IR<14..10>] C ← R[IR<19..15>] − B R[IR<24..20>] ← C			
ADDIS	C	7	R0 = R3 + 15	
	B ← IR<14..00> C ← R[IR<19..15>] + B R[IR<24..20>] ← C			
SUBIS	C	8	R1 = R1 − 3	
	B ← IR<14..00> C ← R[IR<19..15>] − B R[IR<24..20>] ← C			

Instruction Name	Format	Opcode (decimal)	Pseudocode Eq.	Notes
AND	A	9	R3 = R1 & R2	
	B ← R[IR<14..10>] C ← R[IR<19..15>] ∧ B R[IR<24..20>] ← C			
ORR	A	10	R2 = R3 | R2	
	B ← R[IR<14..10>] C ← R[IR<19..15>] ∨ B R[IR<24..20>] ← C			
EOR	A	11	R4 = R0 ^ R3	
	B ← R[IR<14..10>] C ← R[IR<19..15>] ⊕ B R[IR<24..20>] ← C			
ANDI	C	12	R3 = R3 & 4	
	B ← IR<14..00> C ← R[IR<19..15>] ∧ B R[IR<24..20>] ← C			
ORRI	C	13	R2 = R2 | 2	
	B ← IR<14..00> C ← R[IR<19..15>] ∨ B R[IR<24..20>] ← C			
EORI	C	14	R0 = R1 ^ 9	
	B ← IR<14..00> C ← R[IR<19..15>] ⊕ B R[IR<24..20>] ← C			
LDUR	C	15	R0 = &R2[2]	Loads and stores 8 byte numbers
	B ← IR<14..00> C ← R[IR<19..15>] + B MA ← C loadMD R[IR<24..20>] ← MD			
STUR	C	16	&R1[1] = R3	
	B ← IR<14..00> C ← R[IR<19..15>] + B MA ← C : MD ← R[IR<24..20>] storeMD			

Instruction Name	Format	Opcode (decimal)	Pseudocode Eq.	Notes
CBZ	D	17	if (R3 == 0) PC += pointer	
	R[IR<24..20>] = 0 → PC ← R[IR<19..00>]			
CBNZ	D	18	if (R3 != 0) PC += pointer	
	R[IR<24..20>] ≠ 0 → PC ← R[IR<19..00>]			
B	D	19	PC += pointer	
	PC ← R[IR<19..00>]			
BR	D	20	PC += R2	Branches to the location specified by the value of the register
	PC ← R[IR<24..20>]			
B.EQ	D	21	if (Z == 1) PC += pointer	See the flags section, which specifies what the different letters refer to
	Z = 1 → PC ← R[IR<19..00>]			
B.NE	D	22	if (Z == 0) PC += pointer	
	Z = 0 → PC ← R[IR<19..00>]			
B.LT	D	23	if (N != V) PC += pointer	
	N ≠ V → PC ← R[IR<19..00>]			

B.LE	D	24	if (!(Z == 0 &
			  N == V))
			  PC += pointer	

	Z ≠ 0 ∨ N ≠ V → PC ← R[IR<19..00>]			

B.GT	D	25	if (Z == 0 &
			  N == V)
			  PC += pointer	

	Z = 0 ∧ N = V → PC ← R[IR<19..00>]			
B.GE	D	26	if (N == V) PC += pointer	
	N = V → PC ← R[IR<19..00>]			

Instruction Name	Format	Opcode (decimal)	Pseudocode Eq.	Notes
B.MI	D	27	`if (N == 1)` `  PC += pointer`	
	`N = 1 → PC ← R[IR<19..00>]`			
B.PL	D	28	`if (N == 0)` `  PC += pointer`	
	`N = 0 → PC ← R[IR<19..00>]`			
B.VS	D	29	`if (V == 1)` `  PC += pointer`	
	`V = 1 → PC ← R[IR<19..00>]`			
B.VC	D	30	`if (V == 0)` `  PC += pointer`	
	`V = 0 → PC ← R[IR<19..00>]`			
MOVZ	D	31	`R3 = pointer`	Stores the address of the label in the register
	`R[IR<24..20>] ← IR<19..00>`			

Flags

Certain instructions (marked in the instructions list in the notes column) will set flags based on the result. Flags are single-bit registers embedded in the ALU.

N: Negative

The negative flag will be set to *true* if the result of the operation is negative.

Z: Zero

The zero flag will be set to *true* if the result of the operation is `0`.

C: Carry

The carry flag will be set to *true* if a carry occurred out of the most significant bit, or if a borrow occurs on the most significant bit.

V: Overflow

The overflow flag will be set to *true* if the operation overflowed.

Flags impact conditional branch operations. See the instruction list for some examples.

The Simulator

The simulator opens and displays the RAM and CPU in separate windows. The CPU window shows all the registers (both general and special purpose). The RAM window shows the memory of the program, loaded from the file.

Both windows have a refresh button, which forces the display to be re-calculated. This button typically is not needed.

The CPU has two extra buttons, which allow controlling the execution. The Increment Clock button performs a clock cycle and executes a single RTN instruction. The Reset button resets the state of the machine. *Notice: The reset button doesn't reset the memory arrow, which indicates the last memory item to be accessed. The arrow will update appropriately when execution is resumed.*

The simulator **always** reads the file *test_file.o* from the current directory. In the current setup, executing the project from the project directory will function properly. This path can be adjusted in RAM.java, at line 130.

Compiling

```
make compile
```

Running

```
make run
```

Modifications to Original Program

The primary modifications to the program was the addition of the ALU and the implementation of the instructions. Additionally, small changes were made throughout to better improve the program.

The ALU class supports addition, subtraction (with and without setting flags), logical AND, logical OR, and logical XOR. This functionality is strictly contained in the ALU. Furthermore, setting flags sets public single-bit registers in the ALU.

To support the addition of the ALU and the connection between the C register and the ALU, an inheritance hierarchy was created. This allowed functionality in Byte, Bus, and Register to be moved into two classes (BitContainer and BitProvider, where BitContainer isa BitProvider). As ALU is then permitted to function as a BitProvider, polymorphism allows generalized behavior across types.

As a result of this, the Byte and BUS classes are very similar, with the primary different being that the Byte class does not permit variable widths. The Register class has all the functionality of these two classes, with the addition of the idea of sources and destinations.

The implementation of the instructions did not require significant changes to the controller. Some routines were cleaned up, however most provided routines like Fetch0, Fetch1, and Fetch2 were essentially left untouched.

To properly implement most instructions, the word size of the machine was adjusted up to 32 bits. This impacted many areas of the original program which were not created with adjustable word sizes in mind. The current iteration of the CPU should support any word size, however the ISA is designed for 32 bit words.

The memory file loading has also been adjusted. The behavior is unchanged for files which do not use this feature: load each line as a hexadecimal word. However, if desired, the first line can be specified as either `binary` or `hexadecimal.` Using this will cause the appropriate base to be used when reading in a word. The example files use the `binary` option.