Project-3
CSEE 4290

INSTRUCTOR: DR. HERRING
chris.herring@uga.edu

Table of Contents

Architecture Details

Instruction and Address Width

Like ARMv8, instructions have a fixed width of 32-bits, and the addressing space in memory is 32-bits.

Registers

There are 8 internal general-purpose registers, which each take 3 bits to encode, giving a maximum of 9 bits total for register encoding; supporting up to 1 destination register, and up to 2 source registers.

Data Width

32-bit wide data is supported (register and datapath support up to 32 bits), but immediates are limited to 16-bits.

Addressing Modes

There are only 3 addressing modes relevant to most instructions: Immediate Mode, Register Mode, and Register Offset Mode.

Immediate Addressing Mode

Immediate Addressing uses up to a single source register and an immediate value. The result is stored in a single destination register.

Register Addressing Mode

Register Addressing uses up to two source registers and stores the result into a single destination register.

Register Offset Addressing Mode

Register Offset Addressing is the same as Register addressing, with the addition of a signed 16-bit offset to an address in a source register.

Branch Addressing Modes

There are two additional types of addressing for branch instructions, dealing with how an immediate or register address for the branch is treated: Relative addressing and Absolute addressing.

Relative Addressing Mode

Relative addressing is used for the Branch Immediate (B) and Branch Conditional Immediate (B.cond) instructions, and uses the 16-bit immediate field to encode a signed address offset from the current instruction. For example, in the following code snippet, the assembler will generate a 16-bit signed integer to denote a relative address from the branch instruction to the destination of the branch:

```
label:
        ADD R1,R2,R3
        CMP R0,R3,R1
        B.NE label
```

In this case, we'll assume the address of the `B.NE` instruction is `0x1C`, meaning (since instructions are 4 bytes long) the address of `label` is `0x14`. When run, the assembler will generate a relative address as the 16-bit immediate for encoding `B.NE`:

```
0x1C - 0x14 = 8
```

Since `label` comes before `B.NE`, the relative address will be -8, in 2's complement, which is `0xFFF8` in hexadecimal.

Absolute Addressing Mode

Absolute Addressing is used for the Branch Register instruction (BR), and uses a full 32-bit address to denote the next Program Counter location. This allows the programmer to jump much further in their program, with the caveat that you must load a register with the address you wish to branch to before branching.

For example, if you wanted to branch to a subroutine which is outside of the range of a 16-bit signed immediate (between -8192 and 8191 instructions), you would need to use Branch Register, which can branch to any instruction in the addressing space. Say this subroutine is called `printf`, and is defined somewhere in the addressing space:

```
        CMP R1,R2,R3
        B.NE jmpskip
        MOV32 R0, printf
        BR R0

jmpskip:
        ...
        ...

...
>=  8192 instructions later
...

printf:
        ...
        …
```

We first use the `MOV32` instruction to load the full 32-bit address (decoded from label) into register `R0`, then we branch to our function using `BR R0`.

System Flags

The System Flags are N (Negative), C (Carry, Unsigned Overflow), Z (Zero), and V (Signed Overflow).

N (Negative)

Set when an operation in the ALU results in a negative (2’s complement) result.

C (Carry, Unsigned Overflow)

Set when a Carry Out happens in the ALU, signalling an unsigned overflow.

Z (Zero)

Set when the result of an ALU operation is zero.

V (Signed Overflow)

Set when the result of the ALU operation generates a value outside of the allowed size of a signed number in the architecture. Essentially, when a really large number could also be interpreted as a smaller negative number due to the rules of 2's complement.

Error Indication

The input/output model of the CPU includes a 2-bit output for error indication. There are three states for these bits:

- **No Error**: `0b00 (0)`
  - Indicates there is no error or halt, the CPU will continue execution as normal.
- **HALT:** `0b01 (1)`
  - Indicates that the program has ended normally, and halts CPU execution. This also halts the execution of the provided testbench, and prints "Apollo has landed" in the simulation terminal.
- **ERROR**: `0b10 (2)`
  - Indicates that an error has occurred in the CPU, and halts it's execution. We will define these error conditions later in this section. This also halts the execution of the provided testbench, and prints "Houston we have a problem" in the simulation terminal.

Error Conditions

The main purpose of the error condition is to check for undefined instruction encodings which could cause undefined behavior in CPU execution. Most of these will more than likely be caught by the assembler, but having a fail-safe in the CPU ensures no undefined behavior occurs during execution (as long as the CPU design is correct).

Since this architecture is relatively simple, there are not an abundance of options for instructions, meaning less checking needs to be done to ensure an instruction can be executed with a certain option. This leaves the error conditions to mostly be when an instruction encoding is **undefined**, or when a particular combination of bits has no valid instruction to represent.

For example, let's say the assembler erroneously outputs the following opcode:

```
0x2E 08 00 02 = 0b00101110 00001000 00000000 00000010
```

Upon initial inspection, this looks like a regular Data Processing Immediate instruction, as defined in our Instruction Overview section. The instruction encoding looks fine until you get to bits 27 - 25, which are the ALU operation commands:

```
[27:25] = 111
```

If we look at the ALU Operation Commands Table we can see that this ALU Operation Command is **undefined** and could potentially cause undefined behavior in our ALU.

This is a situation in which we'd want to throw an error in the CPU. All of the error conditions can be discovered and accounted for in this manner; simply look for situations in the Instruction Set Architecture where behavior is undefined, and bar that from happening.

Environment Setup

First, clone this repository to your computer using git. Make sure you clone instead of download, so that you can easily track your changes using git.

Then, make sure you have Icarus Verilog and GTKwave installed via the Getting Started instructions for this class.

Then, make sure you have a version of Python 3 installed on your computer. You can use the official Python website for installation instructions.

Next, make sure you have make installed on your system. For Linux users, it should already be installed. Instructions for MacOS and Windows are below:

Make for Windows

To install make on Windows systems, you'll need to first have either MSYS2 (recommended) or Windows Subsystem for Linux (WSL) installed. For instructions on installing and using MSYS2/WSL, please see the instructions included with the Lab 4 files in the Labs repository.

Once you have MSYS2/WSL setup properly by following the instructions in Lab 4, we can begin to install our requirements.

MSYS2 Instructions

Open up an MSYS2 terminal and enter the following command to install make:

```
pacman -S make
```

Test this installation by running the following command in a normal command prompt or PowerShell:

```
make --version
```

If you get some boilerplate text and a version number, you're all set!

WSL Instructions

Open up a WSL terminal and enter the following command to make sure make is installed:

```
sudo apt install gcc gdb make
```

After entering your WSL password and following the prompts, ensure everything installed correctly by executing this command:

```
make --version
```

Make for MacOS

To make sure we have make installed on MacOS, we'll need to use the homebrew package manager. If you don't already have homebrew installed on your MacOS machine, install it using this command:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/insta
ll.sh)"
```

Then, make sure you're package list is updated:

```
brew update
brew upgrade
```

Now, install make:

```
brew install make
```

and ensure that everything worked:

```
make --version
```

Now you should be all set to use this repo!

How to Use the Repository

Now that you have your dependencies setup, you can begin to use this repo for developing your single cycle computer!

In the repository, you'll find a couple of directories:

`parser` contains the `assembler.py` and `instructions.json` files, which are used to convert your assembly code to a '.mem' file, which is readable by the testbench, and consequently your CPU.

`scc` contains the testbench file under the `tests` folder, and a small template for your Single Cycle Computer (`cpu.v`) under the `src` folder.

`tests` contains some sample test assembly files for you to use. They should all assemble correctly and be useful as a test for your Single Cycle Computer.

There is also a document called `scc.docx` which contains a load of more documentation about the class instruction set architecture, the project, and how to use the various tools and templates provided in this repository.

Generally, you'll read through the documentation in `scc.docx` and use that to construct and test your Single Cycle Computer. Your verilog for your Single Cycle Computer should live in `scc/src`, and your test assembly code should live in the `tests` folder.

As an example, to test and run your Single Cycle Computer with a particular test assembly file, execute the following command from the main repo directory. Note, your “python3” executable may differ depending on your system.

```
python3 parser/assembler.py tests/test.asm
```

where `<assembly file>` is the assembly test file you wish to run. Then, change directory into the `scc` folder and execute the following commands:

```
make build
```

```
make simulate ../parser/output.mem
```

which will compile the verilog in the `scc/src` folder and the `testbench.v` file with `testbench` being the top-level module. It will also simulate with the given `output.mem` file, and export that to a file called `dump.lx2` in the `scc` folder. You can view the simulation using `gtkwave`:

```
gtkwave dump.lx2
```

Instruction Overview

Instruction	Type	Instruction	Type
MOV	Data Imm/Reg	B	System/Branch
MOVT	Data Imm/Reg	B.cond	System/Branch
LOAD	Load/Store	BR	System/Branch
STOR	Load/Store	HALT	System/Branch
ADD	Data Imm/Reg	LSL	Data Imm
ADDS	Data Imm/Reg	LSR	Data Imm
SUB	Data Imm/Reg	CLR	Data Imm
CMP [SUBS]	Data Imm/Reg	SET	Data Imm
AND	Data Imm/Reg	NOP	System/Branch
ANDS	Data Imm/Reg		
OR	Data Imm/Reg		
ORS	Data Imm/Reg		

XOR	Data Imm/Reg		
XORS	Data Imm/Reg		
NOT	Data Imm/Reg		

Instruction Encoding

31-30	29	28-25	27-25	24-21	24-22	21-19	18-16	15-0
1LD	**S**	**2LD**	**ALU OC**	B.cond instr	destination reg	op 1 reg	op 2 reg	*16-bit immediate

*16 bit immediate is always signed

1LD: First Level Decode

Binary	Definition
11	SYS / BRANCH
10	LD/ ST
01	Data | Register
00	Data | Immediate

S: ALU/ Special encoding for data instructions**

- Only used for data instructions

Binary	Definition
1	ALU functions
0	Special Functions

2LD: Second Level Decode

- Data / ALU Functions
    - Bit 28 is “set flags”, which sets the “NCZV” flags with the ALU result
    - Bits 27-25 contain the ALU Operation Encoding
- SYS / BRANCH
    - General 2nd level encoding
- LD / ST
    - General 2nd level encoding

ALU OC: ALU Operation Commands

Binary	Definition
001	Add
010	Sub
011	And
100	Or
101	Xor
110	Not

Instructions

Load/Store Instructions

These instructions operate with addresses to access (load to and store from) memory and use Register Offset addressing.

Name	Description	op_code	Instruction	instr (bits)
LOAD	Loads value from memory	load	r_load; o_load	1000000
STOR	Store value of target register.	stor	stor	1000000

Syntax

<mnemonic> <Rd>, <Rp>, #<Offset>

Where <mnemonic> is the instruction mnemonic, <Rd> is the register to load to or store from, <Rp> is the register containing the memory address (the pointer), and <Offset> is a 16-bit signed offset for the address (pointer).

LOAD

Loads value from memory at the location of the address in the pointer register (+/- offset) into the target register.

args

- r_load
  - Reg: [24, 22]
  - Reg: [21, 19]
- o_load
  - Reg: [24, 22]
  - Reg: [21, 19]
  - Imm: [15, 0]

31	30	29	28	27	26	25	24-22	21-19	18	17	16	15-0
1	0	x	x	x	x	0	[Destination Register]	[Pointer Register]	x	x	x	[Offset]

STOR

Stores value of target register to the location in memory of the address in the pointer register (+/- offset).

args

- stor
  - Reg: [24, 22]
  - Reg: [21, 19]

31	30	29	28	27	26	25	24-22	21-19	18	17	16	15-0
1	0	x	x	x	x	1	[Source Register]	[Pointer Register]	x	x	x	[Offset]

Data Immediate Instructions

These instructions operate with/on data directly, using an **immediate** value for at least one operand and register values for other operands.

Name	Description	op_code	Instruction	instr (bits)
MOV	Moves 16-bit imm to lower 2 bytes.	mov	mov	0000000
MOVT	Moves 16-bit imm to upper 2 bytes.	movt	movt	0000001
ADD	Adds imm value and op reg value.	add	i_add	0010001
ADDS	Same as ADD but sets flags.	adds	i_adds	0011001
SUB	Subtracts imm value from op 1.	sub	i_sub	0010010
CMP	Same as SUB but sets flags. (SUBS)	subs	i_subs	0011010
AND	Logical and of op reg and imm.	and	i_and	0010011
ANDS	Same as AND but sets flags.	ands	i_ands	0011011
OR	Logical or on op register and imm.	or	i_or	0010100
ORS	Same as OR but sets flags.	ors	i_ors	0011100
XOR	Logical xor on op register and imm.	xor	i_xor	0010101
XORS	Same as XOR but sets flags.	xors	i_xors	0011101
LSL	Left shift register by an immediate.	lsl	lsl	0000100
LSR	Right shift register by an immediate.	lsr	lsr	0000101
CLR	Clears contents of target register.	clr	clr	0000010
SET	Sets all bits of the target register.	set	set	0000011

Syntax

<mnemonic> <Rd>, <Rs>, #<Immediate>

Where <mnemonic> is the instruction mnemonic, <Rd> is the destination register, <Rs> is the Operand 1 or 'Source' register for the operation, and <Immediate> is a 16-bit signed immediate used as Operand 2 of the operation. <Immediate> is typically in hexadecimal with the syntax #0x<value>; however, shift values are given in decimal (just #<value>).

MOV

Moves a 16-bit immediate value into the lower 2 bytes of the target register. <Rs> is omitted from the instruction syntax.

```
args
```

- `Reg: [24, 22]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-16	15-0
0	0	0	x	0	0	0	[Destination Register]	x	[Immediate]

MOVT

Moves a 16-bit immediate value into the upper 2 bytes of the target register. <Rs> is omitted from the instruction syntax.

```
args
```

- `Reg: [24, 22]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-16	15-0
0	0	0	x	0	0	1	[Destination Register]	x	[Immediate]

ADD

Adds the values of the operand register and immediate value and stores this to the destination register.

```
args
```

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	0	0	0	1	[Destination Register]	[Op 1 Register]	x	[Immediate]

ADDS

Adds the values of the operand register and immediate value and stores this to the destination register - sets flags.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Imm: [15, 0]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	1	0	0	1	[Destination Register]	[Op 1 Register]	x	[Immediate]

SUB

Subtracts the immediate value from operand 1 and stores this to the destination register.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Imm: [15, 0]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	0	0	1	0	[Destination Register]	[Op 1 Register]	x	[Immediate]

CMP [SUBS]

Subtracts the immediate value from operand 1 and stores this to the destination register - sets flags. Works similarly to a compare function.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Imm: [15, 0]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	1	0	1	0	[Destination Register]	[Op 1 Register]	x	[Immediate]

AND

Performs a logical and on the operand register and the immediate and stores the result in the destination register.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	0	0	1	1	[Destination Register]	[Op 1 Register]	x	Immediate

ANDS

Performs a logical and on the operand register and the immediate and stores the result in the destination register - sets flags.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Reg: [15,  0]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	1	0	1	1	[Destination Register]	[Op 1 Register]	x	Immediate

OR

Performs a logical or on the operand register and the immediate and stores the result in the destination register.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	0	1	0	0	[Destination Register]	[Op 1 Register]	x	Immediate

ORS

Performs a logical or on the operand register and the immediate and stores the result in the destination register - sets flags.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	1	1	0	0	[Destination Register]	[Op 1 Register]	x	Immediate

XOR

Performs a logical xor on the operand register and the immediate and stores the result in the destination register.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	0	1	0	1	[Destination Register]	[Op 1 Register]	x	Immediate

XORS

Performs a logical xor on the operand register and the immediate and stores the result in the destination register - sets flags.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	1	1	1	0	1	[Destination Register]	[Op 1 Register]	x	Immediate

LSL

Left shifts the shift register by the value in immediate and stores it in the destination register.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	0	x	0	0	1	[Destination Register]	[Shift Register]	x	Immediate

LSR

Right shifts the shift register by the value in immediate and stores it in the destination register.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Imm: [15,  0]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	0	0	x	1	x	1	[Destination Register]	[Shift Register]	x	Immediate

CLR

Clears the entire contents of the register. <Rs> and <immediate> are omitted from the instruction syntax.

`args`

- `Reg: [24, 22]`

31	30	29	28	27	26	25	24-22	21-0

0	0	0	x	0	1	0	[Destination Register]	x

SET

Sets all bits of the entire contents of the target register. <Rs> and <immediate> are omitted from the instruction syntax.

`args`

- `Reg: [24, 22]`

31	30	29	28	27	26	25	24-22	21-0
0	0	0	x	0	1	1	[Destination Register]	x

Data Register Instructions

These instructions operate with/on data directly, using an immediate value for at least one operand and register values for other operands.

Name	Description	op_code	Instruction	instr (bits)
ADD	Adds imm value and op reg value.	add	r_add	0110001
ADDS	Same as ADD but sets flags.	adds	r_adds	0111001
SUB	Subtracts imm value from op 1.	sub	r_sub	0110010
CMP	Same as SUB but sets flags. (SUBS)	subs	r_subs	0111010
AND	Logical and of op reg and imm.	and	r_and	0110011
ANDS	Same as AND but sets flags.	ands	r_ands	0111011
OR	Logical or on op register and imm.	or	r_or	0110100
ORS	Same as OR but sets flags.	ors	r_ors	0111100

XOR	Logical xor on op register and imm.	xor	r_xor	0110101
XORS	Same as XOR but sets flags.	xors	r_xors	0111101
NOT	Logical not on op register and imm.	not	r_not	0110110

Syntax

<mnemonic> <Rd>, <Rop1>, <Rop2>

Where <mnemonic> is the instruction mnemonic, <Rd> is the destination register, <Rop1> is the Operand 1 register for the operation, and <Rop2> is the Operand 2 register for the operation.

ADD

Adds the values of the operand registers and stores this to the destination register.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Reg: [18, 16]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	0	0	0	1	[Destination Register]	[Op 1 Register]	[Op 2 Register]	x

ADDS

Adds the values of the operand registers and stores this to the destination register - sets flags.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Reg: [18, 16]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	1	0	0	1	[Destination Register]	[Op 1 Register]	[Op 2 Register]	x

SUB

Subtracts operand 2 from operand 1 and stores this to the destination register.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Reg: [18, 16]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	0	0	1	0	[Destination Register]	[Op 1 Register]	[Op 2 Register]	x

CMP [SUBS]

Subtracts operand 2 from operand 1 and stores this to the destination register - set flags. Works similarly to a compare function.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Reg: [18, 16]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	1	0	1	0	[Destination Register]	[Op 1 Register]	[Op 2 Register]	x

AND

Performs a logical and on the operand registers and stores the result in the destination register.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Reg: [18, 16]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	0	0	1	1	[Destination Register]	[Op 1 Register]	[Op 2 Register]	x

ANDS

Performs a logical and on the operand registers and stores the result in the destination register - sets flags.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Reg: [18, 16]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	1	0	1	1	[Destination Register]	[Op 1 Register]	[Op 2 Register]	x

OR

Performs a logical or on the operand registers and stores the result in the destination register.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Reg: [18, 16]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	0	1	0	0	[Destination Register]	[Op 1 Register]	[Op 2 Register]	xx

ORS

Performs a logical or on the operand registers and stores the result in the destination register - sets flags.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Reg: [18, 16]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	1	1	0	0	[Destination Register]	[Op 1 Register]	[Op 2 Register]	x

XOR

Performs a logical xor on the operand registers and stores the result in the destination register.

args

- Reg: [24, 22]
- Reg: [21, 19]
- Reg: [18, 16]

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	0	1	0	1	[Destination Register]	[Op 1 Register]	[Op 2 Register]	x

XORS

Performs a logical xor on the operand registers and stores the result in the destination register - sets flags.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`
- `Reg: [18, 16]`

31	30	29	28	27	26	25	24-22	21-19	18-16	15-0
0	1	1	1	1	0	1	[Destination Register]	[Op 1 Register]	[Op 2 Register]	x

NOT

Stores the 1's complement of the operand into the destination register.

`args`

- `Reg: [24, 22]`
- `Reg: [21, 19]`

31	30	29	28	27	26	25	24-22	21-19	18-0
0	1	1	0	1	1	0	[Destination Register]	[Operation 1 Register]	x

System/Branch Instructions

System and Branch instructions. Includes conditional/unconditional branching, no-op, and other special instructions for the CPU.

Name	Description	op_code	Instruction	instr (bits)
B	Branches to offset defined by imm.	b	i_b	1100000

B.cond	Branches if condition flag permits.	b	b.	1100001
BR	Branches by value in target register.	br	r_br	1100010
NOP	Does nothing, literally!	nop	nop	1100100
HALT	Sets a flag to stop the CPU.	halt	halt	1101000

B

Branches to an offset defined by the immediate (signed).

Syntax

B <label>

OR

B #0x<immediate offset>

`args`

- `Immediate offset:[15, 0]`

31	30	29	28	27	26	25	24-16	15-0
1	1	x	0	0	0	0	x	Immediate

B.cond

Branches to an offset defined by the immediate (signed) only if the proper condition flags are set.

Syntax

B.cond <label>

OR

B.cond #0x<immediate offset>

`args`

- `cond: [24, 22]`
- `Label:[15, 0]`

31	30	29	28	27	26	25	24-21	20-16	15-0
1	1	x	0	0	0	1	[Condition Flags]	x	Immediate

Condition flags:

Encoding	Name	Meaning	Flags
0000	EQ	Equal	Z==1
0001	NE	Not Equal	Z==0
0010	HS	Unsigned Higher or Same (Carry Set)	C==1
0011	LO	Unsigned Lower (Carry Clear)	C==0
0100	MI	Minus (Negative)	N==1
0101	PL	Plus (Positive or Zero)	N==0
0110	VS	Overflow Set	V==1
0111	VC	Overflow Clear	V==0
1000	HI	Unsigned Higher	C==1 && Z==0
1001	LS	Unsigned Lower or Same	!(C==1 && Z==0)
1010	GE	Signed Greater Than or Equal	N==V
1011	LT	Signed Less Than	N!=V
1100	GT	Signed Greater Than	Z==0 && N==V
1101	LE	Signed Less Than or Equal	!(Z==0 && N==V)

1110	AL	Always	Always
1111	NV	Always	Always

BR

Branches to an address defined by the value in a pointer register plus an optional 16-bit signed offset.

Syntax

BR <Rp>, #0x<immediate offset>

args

- Reg: [24, 22]
- Off: [15, 0]

31	30	29	28	27	26	25	24-22	21-16	15-0
1	1	x	0	0	1	0	[Pointer Register]	x	Offset

NOP

Does nothing, literally!

31	30	29	28	27	26-0
1	1	x	0	1	x

HALT

Sets a flag to stop the CPU.

31	30	29	28	27-0
1	1	x	1	x

Assembler Directives/Pseudo Instructions

Assembler Directives and Pseudo Instructions are mnemonics which are not actually unique instructions in the Instruction Set Architecture, but do implement otherwise useful functionality for the programmer. More specifically, **Assembly Directives** are exactly as they sound: directives for the assembler to execute a particular task such as allocating memory, setting the location of a list of instructions in memory, and other related things. **Pseudo Instructions** are aliases of common instruction combinations, such as MOV32 (MOV and MOVT) to load a 32-bit immediate into a register.

ORG

ORG is used to tell the assembler where in memory to write the following instructions to. For instance, if you want to store some instructions or data elsewhere in the **.mem** file, you would use the ORG directive as below:

Syntax

ORG #0x<address>

Where <address> is the address to start writing at.

`args`

- `Imm:[31, 0]`

It is important to note that you **MUST** set the beginning of your program memory to address **0**, so that the CPU will be able to execute your code immediately (the CPU always starts execution at PC = 0).

For instance, if you saved some data or other instructions at memory address **@0000D500** using the directive `ORG #0xD500` at the beginning of your assembly file, then you **MUST** reset the address pointer to 0 before starting your instruction data using `ORG #0x0000`.

Additionally, if you started your instruction memory at the beginning of the file, and followed it with your data at memory address **@0000D500** using `ORG #0xD500`, you would not need to reset the address pointer to 0, as you've already written your instruction memory to the file at address 0.

MOV32

MOV32 is used to move a 32-bit immediate value into a register. This pseudo-instruction converts to a MOV and MOVT instruction, with the lower 16-bits of the immediate being loaded in the MOV instruction, and the upper 16-bits loaded in the MOVT instruction.

Syntax

MOV32 #0x<32-bit immediate>

args

- *Imm:[31, 0]

* Note: this is encoded as 2 instructions, each using one 16 bit immediate

31	30	29	28	27	26	25	24-22	21-16	15-0
0	0	0	x	0	0	1	[Destination Register]	x	Immediate

RMB

RMB reserves 4 bytes of memory for data and skips storing an instruction to this address. Note that you cannot give this location a label, so make sure to keep track of its address (preferably by using the ORG command)

Syntax

RMB

args

None

FCB

FCB reserves 4 bytes of memory for a constant data byte. Note that you cannot give this location a label, so make sure to keep track of its address (preferably by using the ORG command)

Syntax

FCB #0x<32-bit immediate>

args

*Imm:[31, 0]

Parser

Using the Parser

The parser is a python program that converts assembly language written according to the given ISA into hexadecimal machine code that can be run on the verilog testbench or a compliant microarchitecture. The parser is designed as a command line tool that can be called with the python 3 interpreter with the target assembly file passed as an argument. For example, the test assembly file “BabyTest.asm” can be parsed by navigating to the main repo folder in a terminal and executing the command:

```
python3 parser/assembler.py tests/BabyTest.asm
```

An output.mem file and an output.lst file are created in the parser directory. The .mem file contains the machine code ready to be passed to verilog, and the .lst file is a listing file containing the address, machine code instruction, and mnemonic on each line for human reading and debugging.

*In parser, the 16 bit immediate is always in hex.

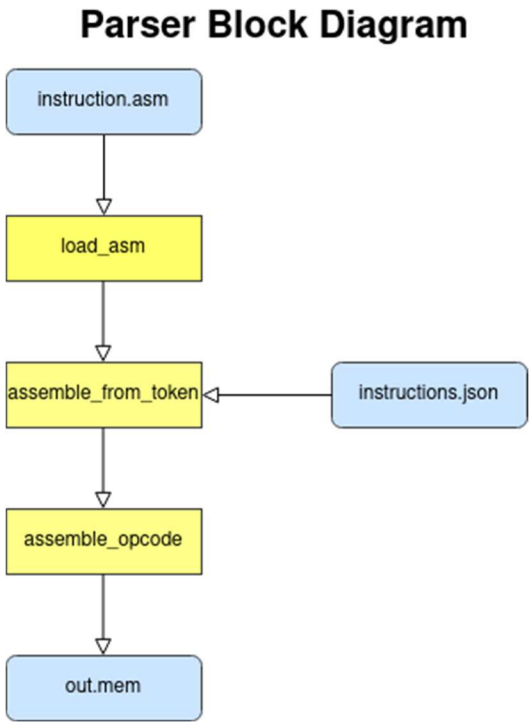
Writing Valid Assembly

- Labels occupy their own line. A valid label begins with a letter and ends with a colon. The assembly within the label block begins on the next line
- Any line that begins with white space (tabs or spaces) will be parsed as an instruction. Arguments are then separated by commas.
- Any text after a semicolon will be parsed as a comment.
- Registers are parsed in the format ‘r2’ or ‘R4’ with the number being the register number
- Immediate values are expected in hex and should be denoted by ‘#0x<value>’
- Empty lines are removed

Valid assembly example:

```
This:
      LOAD    R11, R10
      STOR    R0, R2
      MOV     R0, #0x0; This is a comment!!!
That:
      B.eq    This, #0x20
      NOP
      HALT
```

Parser Operation



Adding to instructions.json

Instructions.json is formed by nested list and dictionary objects and can be easily modified to incorporate new instructions into the parser. First, the instruction is specified, then the opcode is specified for decoding, then the arguments are specified in order they are to be given - with the bits they occupy within the instruction, and finally the opcode bits are provided. Any instruction that follows this general format can be added without modification to the parser program itself.

To add another instruction, navigate to instructions.json and follow the format below:

```
"<Instruction>": {
        "op_code": "<op_code>",
        "args": [
            {"Flg": [index1,index2]},
        ],
        "instr": "<bits>"
```

Example:

```
"b.cond": {
        "op_code": "b.cond",
        "args": [
            {"Flg": [24,21]},
            {"Imm": [15,0]}
        ],
        "instr": "1100001"
```

Assembler

The assembler, assembler.py, reads instructions from the instructions.json file. This file is a dictionary, containing a comprehensive list of instructions, their opcodes, arguments, and mnemonic. The arguments list for each primitive needs to be in the same order as the instruction arguments listed above. The arguments from each line of the assembled file are compared to the instruction primitives from the instructions.json file; if the length and order of the list match, the assembler will begin constructing the instruction based on the selected primitive.

The instruction starts out as zero and the assembler uses shifting and bitwise or operations to construct the instruction. The first step in the construction is adding the opcode; since it is the first seven bits, no shifting is needed. After the opcode, arguments are added in the order they appear in the assembly file; flags require a four-bit shift, registers require three bits. The immediate shift and the label's relative address shift are calculated to make them use bits 15 to 0. Instructions and labels use the same bit locations so the instructions will not have both. The completed instruction for each line is added to a list to be written to a file after all the instructions are constructed. During the writing phase of the assembler, a .mem file and a .lst file are created for running and to facilitate debugging.

Testbench

Testbench Operation

The testbench is a Verilog program used to define data and instruction memory and is used to test and verify the correctness of a program using simulation. Below is an overview of the testbench and its operation.

CLOCKS

The testbench contains three clock signals: *main_clock*, *mem_clock*, and *core_clock*. The *main_clock* signal is the clock running at the simulation tick. The *core_clock* signal is the clock feeding into the *cpu.v* file. The *mem_clock* signal is the clock feeding into memory in the testbench. The *core_clock* is set to never be faster than the *mem_clock* in order to prevent data loss.

INSTRUCTION & DATA MEMORY

The testbench contains separate instruction and data memory, which are defined as 2^15-1 byte locations of memory. The signal *instruction_memory_en* controls whether the instruction memory should be read. If read, the instruction memory at a given address is loaded to *instruction_memory_v*. The signals *data_memory_read* and *data_memory_ write* control whether the data memory is being read from or written to for a given data memory address. Data memory is read on the positive edge of *mem_clock*. Instruction memory is also read on the positive edge of *mem_clock* and on the negative edge of *nreset* (*nreset* signal is low). A NOP instruction is inserted if the *nreset* signal is low.

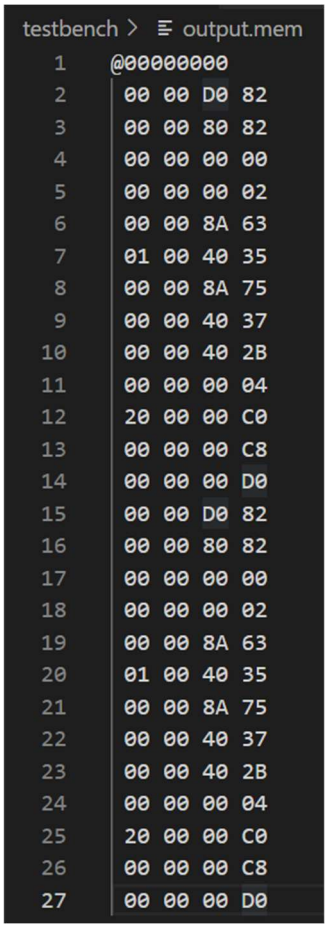
Upon loading the testbench with the chosen '.mem' file, both the instruction memory and data memory will be loaded with the contents of the 'mem' file. This means that initially, your instruction and data memory will be exactly the same, with the only difference being that you can write to the data memory and change its contents at any time.

ERROR INDICATION

If *error_indicator* = 1, this indicates that a HALT instruction was detected. This will halt the execution of the testbench and print "Apollo has landed" in the simulation terminal. If *error_indicator* = 2, this indicates that an error has occurred in the CPU. This will halt the execution of testbench and print "Houston we have a problem" in the simulation terminal. More details on error indication and error conditions can be found in ***Architecture Details***.

.MEM FILE FORMAT

Addresses should be written in the format @00000000. There should be one instruction per line, and each instruction should be written in hexadecimal bytes separated by whitespace. The file *output.mem* can be found below as an example. The parser will output the '.mem' file format correctly, as below, however if you wish to make manual changes to the '.mem' file, make sure to follow the correct format.



Using the Testbench

Below are instructions on how to compile and simulate the *output.mem* file using the testbench:

Open the command prompt and navigate to the *scc* folder.

To compile the testbench, use the following command:

```
make build
```

This should create a file called *testbench.vvp*.

To simulate, use the command:

```
make simulate output.mem
```

This should create a file called *dump.lx2*.

To view the waveform, use the command:

```
gtkwave dump.lx2
```

This will open GTKWave, and signals can be added to view their waveforms.

To remove *testbench.vvp,* use the command:

```
make clean
```

In order to view specific memory locations in the waveform, signals will need to be instantiated within the testbench. To do this, create a wire and assign this to the memory location that is needed to be viewed. An example can be found below:

```
wire [8:0]  mem50;

assign mem50 = memory[80];
```