Computer science, integer RISC instructions (RV64I) and building a hex and macro assembler

Adam Boult (www.bou.lt)

April 29, 2024

Contents

Part I

Arithmetic on states

Chapter 1

Representing natural numbers using states

1.1 States

1.1.1 The bit

A single bit can store a binary piece of information. We can use it to distinguish between two states.

These two states could be represented by True T and False F, but by convention we use 1 and 0.

We can combine bits to store more complex pieces of information. If we have n bits, we can distinguish between 2^n states.

Eight bits together constitute a byte. This can represent one of $2^8 = 256$ states.

1.1.2 Decimal as basis

We could also represent numbers using a decimal basis, so each element could take 10 states, and n elements could represent 10^n states.

The choice of basis is an abstraction.

1.1.3 Representing numbers

By convention (and in particular in C) we can represent numbers using their basis like: $0b0100$ for a 4-bit number, representing 4 in binary. The $0b$ at the start indicates that what will follow is a number written in binary.

Could also write $0d5322$ for 5322, in decimal.

Could write $0x53A4$ for 21412 in hexidecimal.

1.1.4 Endianness

Big-endian: Largest byte in first memory space. Little-endian: Largest byte in last memory space.

1.2 Natural numbers using binary

1.2.1 Representing natural numbers

We will use a byte to describe natural numbers. This gives us a range of $2^8 = 256$. As we include 0 the largest number here is 255.

1.2.2 Zero

We describe zero using all 0s.

$0 = 00000000$

1.2.3 Hexidecimal

4 bits

2^4 is 16

use 0 to 15, 0 to f.

can represent byte with 2 hex. $16x16 = 2^8 = 256$ $0xFA$

Chapter 2

Bitwise operations and the half- and full-adder

2.1 Unary bitwise operations

2.1.1 NOT

We can perform basic operations on inputs.

A simple operation is the unary NOT operator, which returns the reverse of the values of all bits.

We also have binary operators, which take two bits and return another bit. Of note are, AND, OR and XOR.

These operations are a model of Boolean algebra. So there are 16 possible binary functions and 4 possible unary operators.

As in logic, we can combine elementary logical gates to create other logic gates.

These operations can also be performed on a series of bits, however each individual bit is independent of other bits for these operations.

masks in bitwise operations. take only desired part by doing and operation with mask

2.1.2 Setting values

setting values. or operator with target values if want to set to 1. what if want to set 0? invert map and use and. or just use xor

2.1.3 Bitshifts

We can have other operations where bits do interact. An important operator is the bit shift. This takes a series of bits and shifts them to the right or left by one place. This pushes one bit off the end.

The new bit can take a 0 or 1.

Logic gates are not needed for bit shifts. Instead wiring of inputs to outputs achieves the same effect.

shift left is x2 shift right is /2

2.2 Binary bitwise operations

2.2.1 AND, OR and XOR

2.2.2 Combining operations

Simple operators, such as bitwise operators and bit shifts, can be combined to create more complex operators. These could have a large number of inputs, outputs and logical gates.

2.3 Arithmetic Logic Units (ALUs)

2.3.1 Half adder

We first introduce the half adder.

Take two inputs A and B and put both inputs to two binary operators.

The operators are an XOR gate and an AND gate.

The AND gate only returns 1 if both inputs are 1. The XOR gate returns 1 if only one input is 1.

The XOR gate returns the second "digit" while the AND date returns the first.

This means we take two number at either 0 or 1, and return beteen 0 and 2 in binary form.

2.3.2 Full adder

If we are adding, say, a 2 bit number, then we need the abilty to carry numbers. The full adder takes two numbers, and also a carry from the previous digit. The full adder then adds these three numbers.

The carry from each full adder (or half adder) is then passed to the next digit.

If the carry is 1 for the final digit then there has been an overflow.

2.3.3 N-bit ALU

An 8-bit ALU processes numbers represented by 8 bits. Similar for 16-, 32- and 64-bit ALUs.

2.4 Succession

2.4.1 Succession function

We can have a unitary succession function by adding 1 to a number using the full adder.

Chapter 3

Subtraction on natural numbers

3.1 Subtraction and integers

3.1.1 Subtraction

Subtraction works similarly to addition. The equivalent "half adder" returns different values, and the carry forward is not used for addition, but for subtraction.

Chapter 4

Adding and subtracting integers

4.1 Integers

4.1.1 Representing integers

We can expand the natural numbers to the integers.

Consider a byte representing the natural numbers. Previously this would have gone from 0 to 255, with a series of all 1s representing 255.

To introduce integers all numbers with a 1 in the leftmost bit are considered to be negative.

4.1.2 Signed magnitude

You can use the first bit as the sign. eg -1 is represented as 1001.

The downside to this is that arithmetic operations for integers don't work.

4.1.3 Representing using one's complement

If 1 is 0001, -1 is 1110.

One's complement refers to the NOT operation on the binary number.

The advantage of this approach is that it works well with adders.

Downsides are that negative zero exists (0000 and 1111 are both 0).

In addition logic needs to deal with end-around carry.

4.1.4 Two's complement

We can represent the value of negative numbers with two's complement.

In one's complement, the inverse of a number is the NOT operation on the binary number.

For two's complement, the inverse is the number which means $x + inverse(x) = 2^n$ where n is the number of digits.

So we have 0011 as 3, the inverse is 1101 because $3 + 13 = 2^4 = 16$.

The use of two's complement allows us to use the arithmetical logical units for the integers.

4.1.5 Using ALUs with integers

We can expand the natural numbers to the integers.

Consider a byte representing the natural numbers. Previously this would have gone from 0 to 255, with a series of all 1s representing 255.

To introduce integers all numbers with a 1 in the leftmost bit are considered to be negative.

4.1.6 Two's complement

We represent the value of these negative numbers with two's complement. With two's complement the number "after" 127 is -128. Note that this does not just use the first bit as a sign. The use of two's complement allows us to use the arithmetical logical units for the integers.

Chapter 5

Multiplication and division of natural numbers

5.1 Peasant algorithm

5.1.1 Introduction

Bitshifts left for one side, bit shifts right for other.

Part II

Sequential logic

Chapter 6

Sequential logic

6.1 Repeated circuits

6.1.1 Introduction

Can string ALUs together.

6.2 Flip flops

6.2.1 Introduction

Alternative to repeated circuits. Take the result and place it at the input. Needs timer.

Chapter 7

The difference engine

Part III

Finite state machines

Chapter 8

Sequential logic and finite-state machines

8.1 Finite-state machines

8.1.1 Finite-state machines

A finite-state machine has n states and m possible inputs.

Each combination of state and input returns a new state.

This can be representated with an m by n matrix, or a graph.

This is similar to combinatorial logic, but the output is the new state rather than a general output.

This means that successive inputs can be given to a finite state machine, whereas for combinatorial logic only the most recent input matters.

Chapter 9

Regular grammars, regular languages, regular expressions, and Kleene's theorem

9.1 Introduction

9.1.1 Introduction

9.1.2 Kleene's theorem

Equivalence between regular langauges and finite state machines.

Part IV

Pushdown automatons

Chapter 10

The stack and pushdown automatons

10.1 Pushdown automatons

10.1.1 Pushdown automatons

In a finite-state machine the action of the machine depends on the state, and on the input. In a pushdown automaton it also depends on a third input – stack.

In addition, in addition to changing the state, it can also pop the stack, or push to the stack.

The stack is a list of symbols. It is the first item on the stack which is used to inform the decision.

A pushdown automaton uses a Last-in First-out (LiFo) stack.

Chapter 11

Context-free grammar and context-free langauges

11.1 Introduction

11.1.1 Introduction

Chapter 12

The Chomsky hierarchy

12.1 Introduction

12.1.1 Introduction

Part V

Turing machines

Chapter 13

Turing machines

13.1 Turing machines

13.1.1 Turing machines

A Turing machine is a pushdown automaton where the stack is no longer LiFo.

13.1.2 Opcodes

Input is symbol on tape at head, and internal state. Output is:

+ write symbolt to where head is + move left or write or halt. + new internal state

This mapping table is the program. Input to the program can be done by setting the values of the tape at the start.

13.2 Other

13.2.1 Branching

testing for if branches and with target

13.2.2 Instructions

starting cycle. read next instruction pointer. read that instruction. read additional bytes needed. do instruction. wait clock cycles

cpu instructions can vary by required clocks cpu instructions have addressing modeds. they also take addresses. defined address length?

cpu has status register. series of bits about status last instruction etc

13.2.3 Addresses

can address using page and offset. two hex for each. can write eg 0x00FF. last byte of irst page

Chapter 14

Multi-tape Turing machines, Turing equivalence and Turing completeness

14.1 Introduction

14.2 Introduction

14.2.1 Multi-tape Turing machines

14.2.2 Turing equivalence

If two machines can simulate each other they are Turing equivalent.

14.2.3 Turing completeness

If a machine can simulate any Turing machine, it is Turing complete.

Chapter 15

The Church–Turing thesis

15.1 Turing machines

15.1.1 Church-Turing thesis

There is a link between Turing machines, the lambda calculus and general recursive functions. The functions described by each describe the same thing.

Chapter 16

Universal Turing machines, the fetch-decode-execute cycle, and Turing completeness

16.1 Introduction

16.1.1 Universal Turing machines

Rather than have the algorithm be part of the Turing machine, the algorithm is included in the input.

This allows a universal turing machine to simulate any other Turing machine, given the algorithm as an input.

The computer program is stored in memory on the tape.

16.2 Introduction

16.2.1 The fetch-decode-execute cycle

control unit does: + fetch instruction + decode instruction + execute instruction

Regular Turing machines do not need to do this, as there is no program to fetch.

16.3 Turing completeness

16.3.1 Introduction

A computer which can simulate any Turing machine can, by the Church-Turing thesis, compute any effective method on the natural numbers. Such a machine is Turing complete.

As the universal Turing machine can simulate any Turing machine, it is Turing complete.

Part VI

Decidability

Chapter 17

Non-deterministic Turing machines, additional complexity classes (including NP) and complement complexity classes (including co-NP)

17.1 Introduction

17.1.1 Introduction

Deterministic Turing machines don't have complements, they are closed.

Chapter 18

Decidable languages

18.1 Introduction

18.1.1 Introduction

Chapter 19

Reductions

19.1 Introduction

19.1.1 Introduction

Chapter 20

Rice's theorem

20.1 Introduction

20.1.1 Introduction

Chapter 21

Post correspondence problem

21.1 Introduction

21.1.1 Introduction

Chapter 22

Entscheidungsproblem

22.1 Introduction

22.1.1 Introduction

Chapter 23

Semi-decidable languages and recursively enumerable sets

23.1 Introduction

23.1.1 Introduction

Chapter 24

Oracles and Turing reductions

24.1 Introduction

24.1.1 Turing reduction

Treat other problem as a black box, returned from oracle.

If A can be Turing reduced to B, then if there is an algorithm for B, there is an algorithm for A.

If A is Turing reducible to B and B is Turing reducible to A then the problems are Turing equivalent.

Note that this is different to polynomial Turing reduction (Cook reduction).

Chapter 25

The halting problem

25.1 Introduction

25.1.1 Introduction

Chapter 26

Busy beaver

26.1 Introduction

26.1.1 Introduction

Find a halting program of a given size which produces the most output possible.

Describes a Turing machine with an alphabet of 2 symbols.

Part VII

Stack machines

Chapter 27

Stack machines, and their Turing equivalence

27.1 Introduction

27.1.1 Introduction

Extends the pushdown automaton.

To recap, the pushdown automaton had a stack which could be push to, popped, or an operation done on top values on the stack.

Stack machines are Turing complete.

Part VIII

Other Turing-complete abstract machines

Chapter 28

The analytic engine and its Turing-equivalence

Part IX

Register machines

Chapter 29

Counter machines, and program counters (aka instruction pointers) and instruction registers; and Turing-equivalence of counter machines

29.1 Counter machines

29.1.1 Introduction

System has set of registers rather than a tape.

Head of machine points to specific instruction, head position can be changed. Head moves to next line after running (unless a jump)

Instructions include:

+ Increment given register

+ Decrement given register

+ Clear given register

+ Copy contents of one register to another

+ Jump to instruction if a given register is zero

+ If two registers have same value then jump to instruction

29.2 Program counters (aka instruction pointers)

29.2.1 Introduction

program counter: memory address of next instruction to be executed

program counter has address of next instruction. to load puts address in address bus. data bus sends instruction to address bus. data bus sends instruction to CPU

more complex if instruction longer thna 1 byte

The actual instruction is taken from the location stored in the program counter, and stored in the instruction register

29.3 Turing-equivalence of counter machines

29.3.1 Simulating a Turing machine with a counter machine

We know from earlier that a 2 stack machine is equivalent to Turing machine.

A stack can be simulated with 2 counters, therefore a 4 counter machine can simulate a Turing machine.

4 counters can be simulated by 2 counters, and therefore a machine with just 2 counters is Turing equivalent.

29.3.2 Simulating a counter machine with a Turing machine

Chapter 30

Representing machine code with mnemonics

30.1 Bit arrays

30.1.1 Mnemonics

Rather than machine code, programs can be written in human readable form. For example, we can use short English language strings instead of hexidecimal or binary.

mnemonics for op codes

Using literals (for now just integers) in assembly

Chapter 31

Random-access machines and pointers

31.1 Random-access machines

31.1.1 Introduction

Similar to counter machine but:

+ Allows indirect reference of registers. Where r previously would have referred to the register itself, (eg $INC(r)$, we can now refer to the value of a pointer using $[r]$. This allows us to write eg $[3] \rightarrow 4$ which means put the value of register 3 into register 4.

Increments can be rewritten

INC(r) to $[r] + 1 \rightarrow r$ DEC(r) to $[r] - 1 \rightarrow r$

Also for the changes to the instruction register

normally is $[IR] + 1 \rightarrow IR$ with jump is before JZ(r,z). now if $[r] = 0$ then $z \rightarrow IR$. if $[r]! = 0$ then $[IR] + 1 \rightarrow IR$

The register machine introduces additional operations, taking advantage of indirect operations.

31.2 Pointers

31.2.1 Sort

pointers? addresses? as variables? in stack?

31.2.2 Pointers

So in first case we have $x = 2$ and $y = 2$, and then we want to change both to 3.

In first example, initiate both and store 2 in memory twice. update both.

In second example we can have x and y have the same value of a pointer, pointing to a third memory location. we can the update this memory location to change both to 3.

Two variable can be the same by value, but additionally by pointer.

eg $x = 2$ and $y = 2$. $x == y$, but if they have the same pointer, they are the same thing. changing one changes the other.

We can update x and y will automatically update

When we say $x = 2$, $y = x$ we are either making y mutable or immutable. mutable means same pointer. language specific rules apply.

Chapter 32

Random-access stored-program machines

32.1 Introduction

32.1.1 Introduction

Like a random-access machine but stores the program in the registers rather than separately.

This is similar to the relationship between a universal Turing machine and a Turing machine.

Chapter 33

Implementing a stack using a register machine, the frame pointer, the stack pointer, the call stack, return addresses, and the stack buffer overflow

33.1 Introduction

33.1.1 Introduction

Don't need external memory, can just do this in registers.

33.1.2 The stack counter and adding to the stack

33.1.3 Popping the stack

33.1.4 Stack buffer overflow

Chapter 34

Simulating finite state machines with registers

34.1 Introduction

34.1.1 Introduction

Part X

Making RASP machines more realistic

Chapter 35

Limited bit registers - 8-bit, 16-bit etc CPUs

35.1 Introduction

35.1.1 Words

Words (size of register on a pc. eg 8 bits on 8bit, 64 bit on 64 bit)

What does it mean eg to be 8 bit register machine?

16-bit means registers have 16 bits. Can cover 0 to 65535.

8-bit means registers have 8 bits. Can cover 0 to 255.

Each possible value in an address register can refer to a unique word in memory.

35.1.2 Bytes and nibbles

Bytes (8 bit by standard)

Nibbles (4 bit)

35.1.3 How much memory can be accessed by registers

For an n-bit register there can be 2^n possible values, and this can cover $n * 2^n$ bits of data, because each word stores n bits.

For a 8-bit register this means $8 * 2^8 = 2048$ bits, or 256 bytes.

For a 16-bit register this means $16 * 2^16 = 1048576$ bits, or 65536 bytes.

35.1.4 Using hexidecimal to refer to memory addresses

Can refer to memory addresses using $\$FF$ if 8-bit ($16 * 16 = 2^8$).

Can refer to memory addresses using $\$FFFF$ if 16-bit ($16^4 = 2^16$).

35.1.5 Memory extension beyond 2^16 bytes. Memory segmentation

35.1.6 Other

Bit arrays

Bit fields (special use cases for bit arrays? bit arrays more fundamental. bit field more like "we can use first bit for checking if zero"

Chapter 36

External memory, the data bus and the address bus

36.1 Introduction

36.1.1 Using buses to read from and write to external memory

Buses facilitate communication between the CPU (including registers) and memory.

bus takes read (just address) write (address and data)

things can read, rw or write on bus. devices on bus can monitor address outputs of cpu. if address corresponds to device can take actions.

36.1.2 Address bus

Address bus connects to part of memory. setting the address bus to x will allow accessing ram at x.

36.1.3 Data bus

Data bus reads or writes to memory at location of address bus.

eg if loading byte of ram at position $\#FF00$ to accumulator, address bus is $\#FF00$.

Chapter 37

Adding an Arithmetic Logic Unit (ALU) to a CPU

37.1 Introduction

37.1.1 Introduction

We can add an ALU to a register machine. This can give it:

+ Addition. + Subtraction. + Bitmasking. + Bitwise operations.

Part XI

ARM basics

Chapter 38

Advanced RISC Machines (ARM): mnemonics for mov and integer ALU instructions

38.1 Introduction

38.1.1 Introduction

```
mov r0, #0x153
```

makes register 0 hold 0x153

using # indicates an immediate value. value is created by CPU

mov r1, r0 copies r0 to r1

add r2, r0, r1 places addition of r0 and r1 in r2 instead of eg add r0,r0,r1 can write add r0,r1

as well as add can do sub mul and (bitwise) orr (bitwise) eor (exclusive or) lsl (logical left shift) lsr (logical right shift)

Chapter 39

Branches and jumps, loops and branch tables in ARM

39.1 Introduction

39.1.1 Introduction

Chapter 40

External memory, the data bus and the address bus in ARM

40.1 Introduction

40.1.1 Address bus

Address bus connects to part of memory. setting the address bus to x will allow accessing ram at x.

40.1.2 Data bus

Data bus reads or writes to memory at location of address bus.

eg if loading byte of ram at position $\#FF00$ to accumulator, address bus is $\#FF00$.

using RAM: LDR and STR str r0, [r1] store value in r0 at memory location indicated by r1 ldr r0, [r1] load value at r1 in memory to r0 ldr r0, [r1, #4] load value at r1 offset by 4 to r0

Chapter 41

ARM pseudo instruction: ”=”

41.1 Introduction

41.1.1 Introduction

= pseudo instruction

= stuff is shortcut, assember turns it into things with # instead

use of ”=”

ldr r0, =153

Part XII

Implementing functions in ARM

Chapter 42

ARM stack operations

42.1 Introduction

42.1.1 Introduction

Chapter 43

Subroutines in ARM assembly

43.1 Introduction

43.1.1 Introduction

43.1.2 Side-effects

Chapter 44

The Wheeler Jump

44.1 Introduction

44.1.1 Introduction

Used on some machines which didn't have ability to save return address. relevant for registers/stacks?

Before calling the subroutine, put the program counter's current location in the accumulator.

At start of subroutine, take the location in the accumulator, add to it (eg just 1) and write this to the end of the subroutine CODE.

Then when subroutine ends, can go back.

Limitations: slow because writes to memory, not register or stack. also can't do recursion.

Part XIII

ARM assembly

Chapter 45

Assembling assembly code to machine code

45.1 Assembly code

45.1.1 Assemblers

Assembly code can be converted to machine code using an assembler.

The assembler takes the assembly code as input and returns machine code.

Chapter 46

Macros

46.1 Introduction

46.1.1 Introduction

Chapter 47

text, data and bss

47.1 Introduction

47.1.1 Introduction

three parts? + .bss section (block starting symbol) has unitialised statically allocated data + .data section has initialised statically allocated data + code (aka text) which has the instructions *

text file starts eg like

```
section .bss

section .data
  hello: db "Hello world!", 10
  helloLen: equ $-hello

section .text
  global _start
  _start:
```

Part XIV

ARM other

Chapter 48

ARM floating point ALU

48.1 Introduction

48.1.1 Introduction