Specification of the rPeANUt Computer and Assembler (v3.0)

Eric McCreath - 2016
Research School of Computer Science - ANU

Introduction

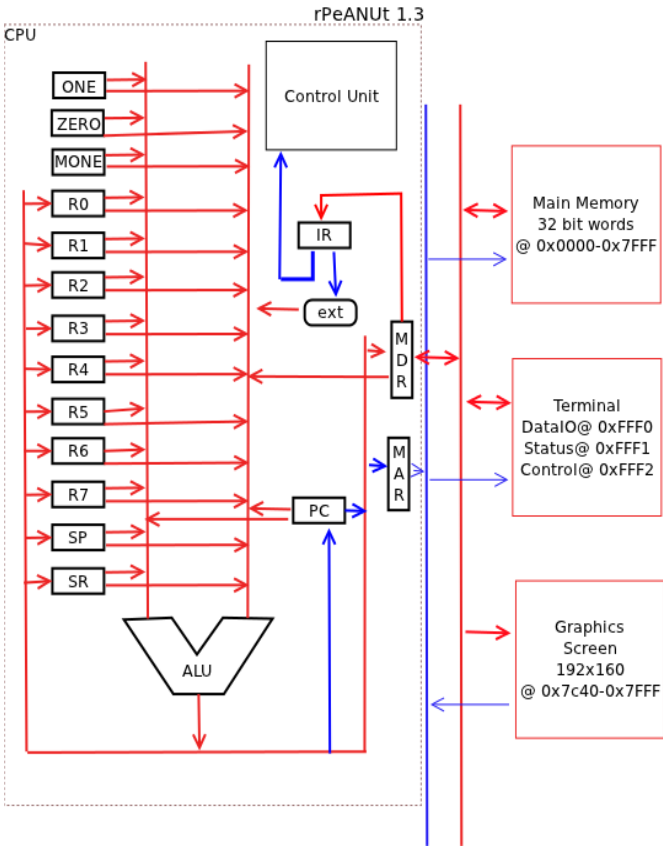
The rPeANUt Computer is a simple microprocessor which was created for teaching computer systems at the ANU. There is a Java implementation of a simulator along with an assembler. This document aims to precisely describe the microprocessor along with the assembler so students can develop code and in so doing gain an idea of what is involved in developing code for a real microprocessor.

rPeANUt Computer - Overview

The rPeANUt is a 32 bit microprocessor with: 16 bit addresses, 32 bit register, and memory that is addressable in words of 32 bits. So the total maximum amount of addressable memory is $2^{16}=65536$ words or 262144 bytes (256k). Only the addresses 0x0000 to 0x7FFF are connected to actual memory. Address between 0x8000 and 0xFFFF are used for memory mapped IO (although only 3 of these addresses are actually used). When the microprocessor is reset the instruction pointer (IP) is set to 0x0100, so normally a program will be load at this point for execution. Addresses 0x0000 to 0x00FF are reserved for the interrupt vector and other OS code. Also the last 960 words of actual memory is used for the frame buffer.

The microprocessor contains the following 32 bit registers:

- 8 generally purpose registers these may be used for storing either data or addresses. These are denoted R0, R1, ... R7.
- An instruction register (IR) - which holds the current instruction that is being executed.
- A status register (SR) - contains status information about the CPU. Bit 0 is used for integer overflow (OF), bit 1 is used for interrupt mask (IM), bit 2 is used for enabling the interrupt timer (TI).
- Three constant registers called ONE, ZERO, and MONE. They contain the constants 1, 0, and -1 respectively.



The microprocessor contains the following 16 bit registers:

- A stack pointer (SP) - this points to the top of the stack and is used for method calls, method returns, and interrupts (SP is set to 0x7000 when the microprocessor is reset).
- A program counter (PC) - which contains the address of the next instruction to execute.

Note that the IR registers is not directly

accessible via the instruction set. Although clearly the execution of instructions will effect this register.

The microprocessor also contains a control unit which sequences the movement of data around the CPU. The microprocessor goes through the follow execution cycle:

```
do {
        IR = mem[PC];
        PC = PC + 1;
        execute_instruction in IR;
        check for interrupts;
} while(!halt);
```

rPeANUt Instruction Set

Instruction are all 1 word long (32 bits). Registers have the labels R0,R1,...R7, SP, SR, PC, ONE, ZERO, MONE and take a nibble (4 bits) in the machine code. The encoding of this nibble is: R0 is 0x0, R1 is 0x1, ... , R7 is 0x7, SP is 0x8, SR is 0x9, PC is 0xA, ONE is 0xB, ZERO is 0xC, and MONE is 0xD. If the register encoding is 0xE then the immediate value within the last 16 bits of the instruction is used. This only works for register sources. Addresses and values take 16 bits of the 32 bit machine code instruction. Values are sign extended from 16 bits to 32 bits. The description in the table below assumes the word of the instruction has been loaded into the IR and the PC has been moved to point to the next instruction word.

Name	**Assembly Instruction**	**Machine code**	**Description**
addition	add <RS1> <RS2> <RD>	0x1<RS1><RS2><RD><value>	RD <- RS1 + RS2
subtraction	sub <RS1> <RS2> <RD>	0x2<RS1><RS2><RD><value>	RD <- RS1 - RS2
multiply	mult <RS1> <RS2> <RD>	0x3<RS1><RS2><RD><value>	RD <- RS1 * RS2
divide	div <RS1> <RS2> <RD>	0x4<RS1><RS2><RD><value>	RD <- RS1 / RS2
modulo	mod <RS1> <RS2> <RD>	0x5<RS1><RS2><RD><value>	RD <- RS1 % RS2
bit and	and <RS1> <RS2> <RD>	0x6<RS1><RS2><RD><value>	RD <- RS1 & RS2
bit or	or <RS1> <RS2> <RD>	0x7<RS1><RS2><RD><value>	RD <- RS1 | RS2
bit xor	xor <RS1> <RS2> <RD>	0x8<RS1><RS2><RD><value>	RD <- RS1 ^ RS2
bit left rotate	rotate <RA> <RS> <RD>	0xE<RA><RS><RD><value>	RD <- RS << RA | RS >>> (32 - RA)
negate	neg <RS> <RD>	0xA0<RS><RD><value>	RD <- - RS
bit not	not <RS> <RD>	0xA1<RS><RD><value>	RD <- ~ RS
copy register	move <RS> <RD>	0xA2<RS><RD><value>	RD <- RS
call immediate	call <address>	0xA300<address>	SP <- SP +1 mem[SP] <- PC PC <- address
return from call	return	0xA3010000	PC <- mem[SP] SP <- SP-1
trap	trap	0xA3020000	SP <- SP +1 mem[SP] <- PC PC <- 0x0002 SR <- SR | (1<<1)
jump	jump <address>	0xA400<address>	PC <- address
jump if zero	jumpz <R> <address>	0xA41<R><address>	if (R == 0x00000000) { PC <- address

Name	Assembly Instruction	Machine code	Description
			}

jump if negative	jumpn <R> <address>	0xA42<R><address>	if ((R&0x80000000) != 0x00000000) {
			PC <- address
			}

jump if not zero	jumpnz <R> <address>	0xA43<R><address>	if (R != 0x00000000) {
			PC <- address
			}

reset status bit	reset <BIT>	0xA50<BIT>0000	SR <- SR & ~(1<<BIT)
set status bit	set <BIT>	0xA51<BIT>0000	SR <- SR | (1<<BIT)
push onto stack	push <RS>	0xA60<RS><value>	SP <- SP + 1 mem[SP] <- RS
pop from stack	pop <RD>	0xA61<RD>0000	RD <- mem[SP] SP <- SP - 1
immediate load from memory	load #<label or value> <RD>	0xC00<RD><value>	RD <- ext(value)
absolute load from memory	load <address> <RD>	0xC10<RD><address>	RD <- mem[address]
indirect load from memory	load <RSA> <RD>	0xC2<RSA><RD>0000	RD <- mem[RSA]
base + displacement load from memory	load <RSA> #<value> <RD>	0xC3<RSA><RD><value>	RD <- mem[RSA + ext(value)]
absolute store to memory	store <RS> <address>	0xD1<RS>0<address>	mem[address] <- RS
indirect store to memory	store <RS> <RDA>	0xD2<RS><RDA>0000	mem[RDA] <- RS
base + displacement store to memory	store <RS> #<value> <RDA>	0xD3<RS><RDA><value>	mem[RDA+ext(value)] <- RS
halt	halt	0x00000000	fetch execution stops!

rPeANUt Hardware and Interrupts

A simple terminal is provided via memory mapped IO. Interacting with this device is done via three addresses: dataIO at 0xFFF0, status at 0xFFF1, and control at 0xFFF2. The least most significant bit of the status register (bit 0) is 1 when data is available and 0 otherwise. Bit 1 of the status register is 0 when it is ready to receive data and 1 if not ready. To write to this device simply write to the memory address of the dataIO location, and to read from this device just read from the dataIO address. Note the status register should be check prior to reading or writing to this device (although good practice to check the status bit before writing, in this simulator it will always be ready for writing, so you may just write directly to the dataIO location). The control address is used to set interrupts on for this device (bit 0 of the control address is 0 if interrupts are disabled and 1 if enabled). If the interrupt bit is set then when the a key is hit an interrupt is generated. Note interrupts are disabled by default.

There is a timer interrupt which is enabled via bit 3 (TI) of the status register (1 enabled, 0 disabled). The timer interrupt will go off when enabled and <clock cycles from start> % 1000 = 0.

The simulated computer has a black and white screen which is 192 pixels wide and 160 pixels high (or 0xC0 wide and 0xA0 high). The contents of this screen is determined by a frame buffer which starts at address 0x7C40 and ends at 0x7FFF. Assuming (0,0) is the top left corner of the

screen then pixel (x,y) will be bit x%32 of the word at address 0x7C40 + 6*y + x/32. If this bit is 1 then the pixel is white and if the bit is 0 then the pixel is black.

When an interrupt occurs the current PC is pushed onto the stack and the PC is set to the address associated with that interrupt. Any registers used by the interrupt service routine must be saved and restored. The interrupt event also sets the interrupt mask high which should be cleared before the interrupt service routine finishes. The standard 'return' instruction is used to return from interrupts. The interrupts and their addresses are given below:

interrupt	address	description
memory fault	0x0000	This happens when memory is accessed that is not addressable.
IO device	0x0001	This happens when interrupts are enabled on the terminal device and a key is hit.
trap	0x0002	This interrupt happens when the trap instruction is executed.
timer	0x0003	When the timer is enabled and every 1000 clock cycles.

rPeANUt Assembler

The rPeANUt assembler provides a simple way of converting assembly code into the rPeANUt machine code. It works using two phases. The first is a line by line translation into machine code. As this translation takes place both a symbol table is created and a list of addresses that need resolving. The second phase involves resolving all these missing addresses. Note the assembler writes directly into the hardware's simulated main memory (this is just for simplicity).

Each instruction must be placed on a single line. Lines with no instructions or labels are simple ignored. Any characters after the first ";" on a line are considered comments and ignored.

As the code is assemble instructions and data are placed into the next available address. The process starts at address 0x0000 and can only move forward. If you wish to skip forward to a new address location then you can simply place the address before a ":". You can not go backwards!

Address labels may consist of alpha numeric characters but must not start with a numeric character. They also must not be keywords (keywords are instructions, register names, and assembler directives). A single address can have multiple names, however, separate lines must be used to achieve this.

Instructions may be placed on a line by them self or after a ":". Instructions have the instruction name followed its parameters. These are all space separated. The names of instructions are given in the table in the instruction section of this document. Please note the order of the instruction parameters is important.

The registers are denoted: R0, R1, R2, … , R7, SP, SR, PC, ONE, ZERO, MONE. Addresses can be given either using an integer (given in base 10 or as hex number, hex numbers are prefixed with 0x) or simply the label. Immediate addresses or values are prefixed with the # symbol. The addressing mode of the load and store operations is determined by the list of parameters.

The "block" keyword is used to reserve a block of memory. If the block keyword then a positive integer *k* is given then *k* words are reserved (these are initialised to 0). If block followed by an immediate integer or character then one word is reserved and this word is initialised to the given value. If a string is given using `#"<string>"` (e.g. `#"Hello World"`) then a block of words is reserved and initialised to that string. The characters are stored in the least most significant byte of each word, and a null terminator is appended.

Constant values can be in decimal, hexadecimal or a character (the character's code is used as the value). For hexadecimal the numbers should start with "`0x`". Characters should be place between two single-quotes, eg. `#'c'` would be equivalent to `#99` or `#0x63`. Escapes also work: `#'\n'`

The “`#include`” keyword can be used to insert source from another file. The first and only argument is a string (without a `#`) specifying the file to include. Note that the file path is relative to the folder you ran rPeANUt in. eg. #include "drawinglib.s"

“#define” may be used to replace a label with a string, this is basic textual replacement and may be used for named constants or to simplify repeated code. e.g.

```
#define start 0x0100
#define setTen load #10

start : setTen R5
```

Simple macros may be defined within the assembly code. This is done using a number of lines of code:

- the first line is the “`macro`” key word,
- the next line is the macro name along with its parameter variables (parameter variables start with the '&' symbol),
- then the lines of code that make up the macro (these may include variables), and
- finally the macro is ended with the “`mend`” on a line.

Macros may be used by the just stating the macro name followed by the parameters. Macro must be defined before they are used and they are just textual replacement directives. The below is an example of use a macro for the hello world program:

```
macro
putc &c
      load #&c R1
      store R1 0xfff0
mend

0x100 : putc 'h'
        putc 'e'
        putc 'l'
        putc 'l'
        putc 'o'
```