Contents

Listings

Chapter 1

Introduction

The Wombat simulator is based on a design by Dale Skrien. The design is an example architecture for his *CPU Sim* simulator. This version was adapted by Dan Bennett with the addition of several instructions.

Chapter 2

Architecture

Memory
- Input/Output
- ALU
- Control Unit

2.1 Hardware Description

The Wombat hardware consists of one general purpose register, called an accumulator, four special purpose registers, a control unit, an ALU (or Arithmetic Logic Unit), and an internal bus. Associated with the CPU is a memory, and an I/O unit. The instruction set consists of 15 instructions, requiring an opcode of 4 bits, with 12 bits of the 16 bit word remaining for operands. This limits the addressable memory to 2^{12} bytes.

Memory is byte addressable, but all instructions must be word aligned. Furthermore all data access is performed on word aligned boundaries on word sized data.

2.1.1 Registers

The register set consists of the accumulator (ACC) , the instruction register (IR), the program counter (PC), the memory address register (MAR), and the memory data register (MDR). Input and output are accomplished through the input register (inREG) and output register (outREG). All registers are 16 bits wide and are initialized to zero. Each register will be discussed

in turn.

The accumulator is only general purpose register available. It is the target for all computations, input and memory loads. The value stored in the accumulator is the left hand source for all arithmetic operations, memory stores and output. The accumulator is initialized to zero. The ACC supports direct transfer to and from the ALU.

The program counter contains the address of the next instruction to be fetched. Initialized to 0, this register supports increment by two.

The instruction register holds the current instruction. This register supports extraction of the opcode or top four (12-15) and, operand or bottom 12 (0-11) bits of a word.

The MAR holds the address for memory access. This is a portion of the memory management unit (MMU). This register can be loaded from either the bus or from the MDR.

The MDR holds data transferred to, or from memory. The MDR can read and write to the bus, but also supports direct transfer to the MAR.

2.2 Other Hardware Components

Wombat's memory is byte addressable, however all transfers are performed at the word level, on word aligned boundaries. This is a big-endian machine, so bits 8through 11 are stored in the even byte and bits 0 through 7 are stored in the odd byte.

Memory files are stored in a format compatible with version 2.0 of Tk-Gate. In this case, each block of values is proceeded by a block header in the form "address". The address is given in hexadecimal. The block header is the only entry on the line. A block of data, in hexadecimal, follows the heading. Data should be words (16 bits) delimited by spaces. A data block ends with a blank line. Figure 2.1 is the compiled version of the example program bigger.

2.3 Register Transfer Notation

Register Transfer Notation (RTN) used for this document employs the following conventions.

Figure 2.1: Memory File for Program Bigger

```
@0
3000 2016 3000 2018 6016 a010 1018 b012 1016 4000
0 0 0
```

- Register names (ACC, PC, IR, MAR, MDR, ...) are used to represent the registers.
- Subscripted registers represent the specified bits in that register. Ir[0..11] represents the operand, bits 0 through 11 of the instruction register.
- The bus (BUS) is treated as a register.
- Memory is represented as the array M. M[0] represents memory location 0.
- The assignment operator $\leftarrow$ moves a value from the location on the right hand side to the location on the left hand side. $ACC \leftarrow BUS$ represents moving the value on the bus to the accumulator.
- Basic arithmetic operations are represented by the following symbols: {+,-,*,/}. A generic operation is specified by *op*. `PC+2` represents adding two to the PC, while `ACC op BUS` represents performing an operation (specified by control codes) on values stored in the accumulator and the bus.
- `Input` reads a 2 byte signed integer from the input device to inREG.
- `Output` writes a 2 byte signed integer from the outREG to the output device.
- Conditional branches are evaluated using the `if (expression) operation` format.

2.4 Instruction Set

Table 2.1 instructions are supported in the enhanced wombat instruction set.

Mnemonic	**OpCode**	**Operand**	**Operand Bits**	**RTN**
STOP	0			Stop Execution
LOAD	1	address	0 through 11	ACC ← M[address]
STORE	2	address	0 through 11	M[address] ← ACC
READ	3			Input ACC ← inREG
WRITE	4			outREG ← ACC Output
ADD	5	address	0 through 11	ACC ← ACC + M[address]
SUBTRACT	6	address	0 through 11	ACC ← ACC - M[address]
MULTIPLY	7	address	0 through 11	ACC ← ACC * M[address]
DIVIDE	8	address	0 through 11	ACC ← ACC / M[address]
JMPZ	9	address	0 through 11	if (ACC == 0) PC ← address
JMPN	A	address	0 through 11	if (ACC < 0) PC ← address
JMP	B	address	0 through 11	PC ← address
ADDI	C	immediate	0 through 11	ACC ← ACC + immediate
LOADI	D	address	0 through 11	ACC ← M[M[address]]
STOREI	E	address	0 through 11	M[M[address]] ← ACC

Table 2.1: The WOMBAT instruction set

Chapter 3

Assembler

The assembler is currently in beta version. It has weak syntax error diagnostic reporting.

The assembler is line oriented and only one instruction is permitted per line. All elements are delimited by white space.

The general form of an assembly line are:

```
[label:] [mnemonic [label | number]] [comment]
label: .data number number [comment]
```

3.1 Comments

Comments begin with either a semicolon (;) or with two slashes (//) and end with the end of the line. As such, a comment must be the last item on a line.

3.2 Numeric Literals (numbers)

The assembler supports signed integer literals in four formats: binary, octal, decimal and hex. Decimal values are the default and are represented as integers. Rules for encoding other values are given in table 3.1

3.3 Mnemonics

The instruction set is given in table 2.1 with Mnemonics listed in the first column. Mnemonics are case insensitive but must be delimited by white

Format	Prefix	Example
Binary	bx	-bx0011001000001001
Octal	ox	ox1734
Decimal		-1234
Hexadecimal	hx	hx3245

Table 3.1: WAS Number Representation

space.

3.4 Labels

Labels are case sensitive, must begin with an alpha character and contain only letters, digits and the underscore. A label must not be a reserved word. Labels can be used in place of any operand. All labels must be resolved in the first pass of the assembler.

In order for a label to be resolved, it must be declared. Labels are declared as an optional first element of a line. A label declaration consists of a legal label followed by a colon. Multiple labels can point to the same address.

3.5 Data

Data is declared with a data line. Such a line must begin with a label, contain the assembly directive `.data`, a size (in bytes) and a value. The byte size must be even. All words are assigned the same value.

The directive:

```
foo: .data 6 -1
```

sets aside three words (6 bytes) at the current memory location. Each word is initialized to -1.

3.6 Invoking the Assembler

The current version of the assembler expects the filename of the source code to be the last argument given. Other arguments are listed in table 3.2.

Examples of command line use include:

Flag	Arguments	Explanation
-o	output file name	The default output file is `a.out`. This flag will override the default and use the next argument as the output file name.
-v		Verbose. A small amount of diagnostic information is printed when the -v flag is used.
-t		Symbol Table. Print the labels encountered in the program along with addresses.

Table 3.2: WAS Command Line Arguments

```
was bigger.a
```

Assembles bigger.a to produce the executable file a.out.

```
was -o bigger bigger.a
```

Assembles bigger.a to produce the executable file bigger.

3.7 Example Code

The following code examples represent the test cases for the was assembler. The source code for these programs in available in the `progs` directory in the WOMBAT source code tree.

Please note, the text processing package may have wrapped some of the lines in the code listing in this document. Comments extending over multiple lines are not legal in WOMBAT assembly.

```
;
; This program will read in two values and print out the
   bigger
;

; read and store the first number
        read
        store a

; read and store the second number
        read
```

```
        store b

; subtract acc = b-a
; if acc < 0 then a> b, else b>= a
        subtract a
        jmpn bigA
;       b is bigger, so load b for output
        load b
        jmp end
;       a is bigger, so load a for output
bigA:   load a

;       print the bigger number
end:    write

;       exit
        stop

;       storage for the numbers
a: .data 2 0
b: .data 2 0
```

Listing 3.1: bigger.a, code to print the larger of two numbers

This program will read two values from the user and print the larger of the two.

```
        load zero         ; initialize the LCV
Loop:
        store LCV         ; store the LCV
        subtract maxsize  ; if lcv >= MAXSIZE
        jmpz  END         ; exit the program
        load LCV          ; tmp = lcv * lcv
        multiply LCV
        write             ;  cout << tmp
        load LCV          ;  lcv ++
        addi 1
        jmp Loop

END:    stop
```

```
zero:           .data 2 0
LCV:            .data 2 0
maxsize:        .data 2 10;
```

Listing 3.2: power.a, code to compute i^2 for $i <= 10$

A simple looping program to compute i^2 for $0 <= i <= 10$. This program demonstrates use of a loop control variable (LCV), the addi and jmpz instructions.

```
; this program will attempt to emulate a random number
   generator
;

        read
        store RNG_seed
        ; i = 0
        load zero
        store i
Loop:
        load i
        subtract max
        jmpz EXIT

        ; seed * a + c
        load RNG_seed
        multiply RNG_A
        add RNG_C
        store RNG_seed

        ; seed % m <-> seed - (seed/m)*m
        divide RNG_M
        MULTIPLY RNG_M
        store RNG_TMP
        load RNG_seed
        subtract RNG_TMP
        store RNG_seed

        // done with RNG
        write
        load i
```

```
        addi 1
        store i
        jmp Loop

EXIT:   stop

; data for the main loop
zero:   .data 2 0
i:      .data 2 0
max:    .data 2 10

; data area for the RNG
RNG_seed:       .data 2 0
RNG_A:          .data 2 41 ;(a-1) should be divisable by
   all prime factors of m
RNG_C:          .data 2 17 ; c and m should be relatively
   prime
RNG_M:          .data 2 100
RNG_TMP:        .data 2 0
```

Listing 3.3: random.a, a program to generate 10 random numbers

This program uses the Linear congenital method for random number generation. ($X_{n+1} \equiv (aX_n + c)\%m$). The program asks the user for a seed value and generates ten random values based upon that seed.

```
; This program will compute the squares of the numbers 0
   through 19
; But it will do it by storing the values in an array and
   then squaring
;     each entry into the array
;
; It will finish by printing the array

; initialize the array
; for(i=0;i<size;i++) {
;     array[i] = i
; }

Loop1:
        load i          ; load lcv
```

```
        subtract size    ; subtract the upper limit
        jmpz Loop1E      ; branch if the two are equal, (
           exit loop)

        load    i        ; load i
        add     i        ; (2i)
        addi array       ; compute the effective address (
           base + 2*i)
        store   address

        load i
        storei  address ; m[address] = i
        addi 1           ; i++
        store i          ; store the new lcv back to i
        jmp Loop1

Loop1E:

; Now step through and square all elements in the array
; for(i=0;i<size;i++) {
;     array[i] *=  array[i]
; }

        load i           ; i = 0
        subtract i
        store i
Loop2:
        load i           ; if i == size
        subtract size
        jmpz Loop2E      ; end loop

        load i           ; address = base + 2*i
        add  i
        addi array
        store address

        loadi address    ; tmp = m[address]
        store tmp        ; tmp = tmp * tmp
        multiply tmp
```

```
        storei address  ; m[address] = tmp* tmp

        load i          ; i++
        addi 1
        store i;
        jmp Loop2

Loop2E:

; finally, print out the array

        load i          ; i = 0
        subtract i
        store i
Loop3:
        load i          ; if i == size
        subtract size
        jmpz EXIT       ; end loop

        load i
        add  i          ; address = base + 2*i
        addi array
        store address

        loadi address   ;
        write

        load i          ; i++
        addi 1
        store i;
        jmp Loop3

EXIT:
        stop

// data section

i:              .data 2 0
tmp:            .data 2 0
size:           .data 2 20
```

```
address:        .data 2 0
array:          .data 40 -1 ; initialize the entire array
    to be -1
```

Listing 3.4: SimpleArray.a, a program to demonstrate array access

This program demonstrates array access.

3.8 Error Messages

Chapter 4

Simulator

There are two version of the simulator. An instruction level based simulator and a microcode level GUI simulation.