Sigma16 User Guide

John T. O'Donnell

Version 3.10.0, June 2025

`https://jtod.github.io/home/Sigma16`

Contents

1 Introduction

Sigma16 is a computer architecture designed for research and teaching in computer systems. This application provides a complete environment for experimenting with the architecture. It includes an editor, assembler, linker, emulator, a collection of examples, a user guide with tutorials, and an integrated development environment (IDE) with a graphical user interface.

1.1 Running Sigma16

Sigma16 runs in a web browser: you don't need to download or install anything.

To run Sigma16, visit the Sigma16 Home Page at `https://jtod.github.io/home/Sigma16` and click the prominent *Run* link.

There are additional command line tools that can run in a shell; see the Installation section. There is also a digital circuit (available separately) that implements the architecture. Using these tools, machine language programs can run on both the emulator and the circuit.

1.2 Browser compatibility

This version of Sigma16 runs on up-to-date versions of browsers based on Chromium, including Opera, Chrome, Edge, DuckDuckGo. Currently, it does not work on Safari or Firefox.

1.3 Organization

The architecture is organized into subsets to make it easier to learn and to use:

- The **Core** subset has a small instruction set which is a good starting point for learning about computer architecture. Although simple, Core is powerful enough to support realistic programming.
- The **Standard** subset offers flexible programming techniques, including manipulation of bits, Boolean expressions, shifting, extracting fields, arithmetic on natural numbers and arbitrary precision integers, and concise support for stack frames. It also supports the study of systems programming, providing interrupts, protection, concurrent processes, mutual exclusion, and memory management.

For a quick start, begin with the Core tutorials, which introduce the architecture step by step. The tutorials explain the machine, show how to program it, and demonstrate how to enter and run a program and how to use the programming environment. A reference section follows the tutorials.

Our focus is on fundamental concepts, ideas and principles. Sigma16 illustrates the basic ideas of computer systems but it avoids unnecessary complexity. For example, Sigma16 has just one word size (16 bits) while most commercial machines provide a variety. That variety is useful for practical applications but it complicates many of the details while not adding new fundamental ideas. Most commercial computers that achieve success in the marketplace evolve through updates and extensions, and eventually become encrusted with complications that help support backward compatibility, leading to complexity.

2 Core architecture tutorials

The following short tutorials introduce the system; full details appear in later sections. You can keep the tutorials visible in the right panel while following along with the examples in the main panel.

2.1 Hello, world!

Let's begin by running a simple example program. For now, we focus only on how to use the software tools. You don't need to understand the example code yet. An explanation of the program and the Sigma16 architecture will come later.

To launch the app, visit the Sigma16 Home Page and click on the link to run it. Sigma16 runs in the browser; you don't need to download or install anything.

- Click **Editor**, then **Hello, world!**. This will enter a small assembly language program into the editor window. Later, we'll load some of the more complex example programs into the editor, and you can also modify a program or type in a new one from scratch. For now, don't worry about the content of the program.

- Click **Assembler**. The assembler translates programs in assembly language (the source program) to machine language (the object program). Assembly language is a human-readable notation, while machine language is what the computer can execute. When you first enter the Assembler tab, the assembly language text in the Editor window is copied over; this is the source program.

- Click the **Assemble** button. This will do the translation. The **assembly listing** is displayed: this shows the original source code along with the machine language, and any error messages. The **Show object** button displays the object code, which is the machine language program produced by the translation. The **Show source** button displays the original source code, and the **Show Listing** button displays the assembly listing again.

- For this simple example, we don't need the Linker, so you can skip it. The linker is needed for larger and more complex programs with multiple modules, or with external references, or that need relocation.

- Click **Processor**, which shows the main components of the computer architecture, including registers and memory. These components are explained later. For now, just note that this page is where you can run programs using the emulator.

- Still on the Processor page, click **Boot**. This reads the machine language program into the memory, and you can see it in the Memory display. The source code (the assembly language) appears in the bottom section. If the assembler produced any error messages, the program will not boot until you fix the errors.

- Click **Step**. The processor executes a single instruction and displays the effects on the registers and memory: blue for *using* a value, and red for *modifying* it. The assembly listing shows the instruction that just executed by highlighting it in red. It also shows the instruction that will execute next by highlighting it in blue. This is just to make it easier to follow what is happening; the machine executes the machine language program, which is in memory, and it ignores the assembly language listing which is just a convenience to help you follow the program. The machine relies only on the registers and memory, and doesn't even "know" that the assembly listing exists.

  There is a keyboard shortcut: after you have clicked Step once, you can press the space bar to execute the next instruction. Thus you can step through the program by clicking Step, and then pressing the space bar repeatedly.

- Click Step repeatedly to watch the program execute, instruction by instruction. When the program terminates, the small window labelled **Emulator** will display **Halted**.

- You can also run the program to completion, without having to click Step so many times. Click **Boot** again to get the machine back into the initial state. Now click **Run**, and the program will continue executing instructions until it halts.

To run the program slowly, click Step repeatedly. To run the program faster but without updating the display after each instruction, click Run. At any time you can click Pause to stop the processor, and you can resume execution with either Step or Run. Sometimes it's useful to let the processor

run at full speed until it reaches a particular instruction, and then stop. This can be done by setting a **breakpoint** (described in the Breakpoint tutorial below).

To exit the app, just close the browser window or tab. This may put up a dialogue box warning that any unsaved data may be lost and asking you to confirm.

2.2 A quick tour

This tutorial introduces the main components of the architecture as well as the graphical user interface.

The main window contains two main sections. The largest area, on the left side, is the **main working area**. When the program launches, this will show the Welcome page. The **user guide** is on the right side. At the top is a row of buttons (Welcome, Examples, etc.). These select which page is displayed in the main working area.

It's convenient to see the main working area and the user guide side by side. Begin by resizing the entire window (bigger is better). Then you can change the amount of space given to the user guide by clicking the arrow symbols on the right side of the top button bar. These arrows will expand or shrink the user guide: the small arrows adjust by one pixel, the larger arrows by ten pixels. If you resize the entire browser window, Sigma16 will maintain the same relative sizes of the main working area and the user guide sections.

If your screen is small, and the main working area isn't big enough, click *Hide User Guide* and all the space will be made available to it. The button will change to *Show User Guide*.

You can also open the User Guide in a separate browser tab or window. The Welcome page contains a link to do this.

The main working area has several pages, with tab buttons at the top to switch between them:

- **Welcome** contains some introductory information and links.
- **Examples** contains a collection of assembly language programs organized by the architecture subset. Start with the Core examples.
- **Modules** shows a summary of all the files and modules you currently have open. It also provides buttons allowing you to open files on your computer, close them, and select one to work on.

- **Editor** shows the selected module, where it can be edited. You can assemble and execute the selected module. To run a program, you'll load it into the Editor (there are several ways to do this), then assemble it (Assembler tab) and then run it (Processor tab).
- **Assembler** translates a program from assembly language to machine language, and shows the assembly listing as well as the object code.
- **Linker** is used to combine a collection of object code modules into a single executable program. It also performs external name resolution and relocation. Programs that consist of just one module do not need the linker, and you can skip it.
- **Processor** shows the components of the architecture and executes machine language programs.
- **Options** allows you to configure how the system operates.
- **About** gives general information about the software, including the version.
- **Hide User Guide** makes the entire window available to the main working area. It is a toggle that changes to **Show User Guide**.

2.3 Registers, constants, and arithmetic

Programs do most of their work using the **register file**, which is an array of 16 registers named R0, R1, R2, . . . , R15. The Register File is displayed in a box on the Processor page.

A register is a circuit that can hold a number, and the elements of the register file can be used to hold variable values. They are analogous to the registers in a calculator: think of each register as a box that can hold a number, and think of the register name as a variable name. Two of the registers, R0 and R15, are special and should not be used to hold variables.

A computer program is a sequence of **instructions**. Instructions are similar to statements in a programming language, but they are simpler.

Sigma16 performs arithmetic on data in registers. To do any computation on some numbers, we first need to get those numbers into registers. The **lea** instruction. can be used to place a constant into a register. For example, to load 42 into register 3, write

```
lea  R3,42[R0]     ; R3 := 42
```

This is a statement in **assembly language**, and it describes one instruction. This statement contains three parts:

- The **operation** is **lea**. This tells the computer what action to perform, and "lea" says to put a value into a register. Later we will see why this instruction is called "lea".
- The **operands** are **R2,42[R0]**. R2 is the *destination*, where the value will be placed. The value to be put into the destination is 42. For now, ignore the **[R0]**; its purpose will be described later.
- Everything after the semicolon is a **comment**. This comment is a programming language statement that describes what the instruction does: it sets the variable R3 to 42.

The name **lea** is the **operation**, i.e. the name of the instruction. The operand field consists of two operands separated by a comma. The first operand, R2, is called the **destination**; this is the register where the result will be placed. The second operand is a constant 42 followed by [R0]. When the computer executes this instruction, it simply places the constant into the destination. In a higher level language, we could write **R2 := 42**.

Most instructions follow a similar pattern: the first operand is the destination where the result is placed, and the subsequent operands are the arguments to the computation. This is the same convention used in assignment statements in many programming languages: the registers in add R1,R2,R3 appear in the same order as the variables in R1 := R2 + R3.

All arithmetic operations take place in the registers, and there is a separate instruction for each operation. For example, the following instruction will add the values in R8 and R1 and then put the result into R4:

```
add   R4,R8,R1  ; R4 := R8 + R1
```

Notice that the operand field doesn't use operators like := or +; those are specified by the operation field (lea, add). Each instruction performs only one operation. The operand field just separates the operands with commas. The first operand (R4 in this example) is the **destination**, which is where

the result will be placed. The last two operands (R8 and R1) are the values that will be added.

To perform a calculation, we need to get the data into registers (using lea) and then perform the calculation (using arithmetic instructions). The following program calculates 3 + 4 and puts the result into R2:

```
lea   R5,3[R0]    ; R5 := 3
lea   R8,4[R0]    ; R8 := 4
add   R2,R5,R8    ; R2 := R5 + R8 = 3+4 = 7
```

It's a good idea to use comments to explain the meaning of an instruction. For now, comments like "R4 := R8 + R1" will be used to show what the instruction does. That's useful while learning what the instructions do, but later on we will use comments to give more meaningful information (for example, what do the values in the registers mean, and why are we adding them?).

There are three more arithmetic instructions. These follow the same pattern as add: in each case, the arithmetic is performed on the last two registers and the result is placed in the destination (the first register):

```
add  R4,R11,R6   ; R4 := R11 + R6
sub  R5,R2,R13   ; R5 := R2 - R13
mul  R2,R10,R7   ; R2 := R10 * R7
div  R5,R6,R12   ; R5 := R6 / R12, R15 := R6 rem R12
```

The divide instruction is slightly different: it produces two results, the quotient and the remainder. The quotient is placed in the destination, and the remainder is automatically placed into R15, even though the instruction doesn't mention R15. If you write **div R15,R1,R2**, the quotient is placed in R15 and the remainder is discarded.

Normally an arithmetic instruction will put a new value into the destination register, but the operand registers are left unchanged. However, what happens if one of the operands is the same as the destination, for example **add R7,R7,R8**?

An arithmetic instruction proceeds in three phases: (1) obtain the values in the operand registers; (2) perform the arithmetic on those values; and (3) put the result into the destination, discarding whatever value was previously

there. So consider this example:

```
lea   R7,20[R0]  ; R7 := 20
lea   R8,30[R0]  ; R8 := 30
add   R7,R7,R8   ; R7 := R7 + R8
```

After the two lea instructions have executed, R7 contains 20 and R8 contains 30. The add instruction does the following:

1. It fetches the values in R7 and R8, obtaining 20 and 30
2. It adds the values, obtaining the result 50
3. It puts the result 50 into the destination R7, discarding the previous value.

The final result is that R7 contains 50.

Constant data can be specified using either decimal or hexadecimal notation.

- Decimal numbers are written as strings of digits, optionally preceded by a minus sign: 3,-19, 42. Leading zeros are optional, so 23 and 0023 both mean the same (twenty three).
- Hexadecimal numbers are written as four hex digits, and in assembly language programs they are indicated by putting $ before the number. Thus $00a5 and 0165 both represent the integer 165.

```
lea   R1,13[R0]     ; R1 =  13 (hex 000d)
lea   R2,\$002f[R0]  ; R2 := 47 (hex 002f)
lea   R3,\$0012[R0]  ; R3 := 18 (hex 0012)
lea   R4,0012[R0]   ; R4 := 12 (hex 000c)
```

The processor page shows numbers as hex without the leading $, but in an assembly language program the $ is needed to avoid ambiguity.

Sigma uses := as the assignment operator; thus we write R7 := R7 + R8 (and we don't write R7 = R7 + R8). This is because an assignment statement is profoundly different from an equation, and mathematicians have long used the = operator to indicate equations. It isn't just an academic

or theoretical point; there have been plenty of occasions where computer programmers get confused between assignment and equality, and using the wrong operator leads to serious bugs. (A major security breach occurred because a C programmer forgot that in C, = does not mean equals.)

Why does assembly language use a notation like add R5,R2,R3 instead of R5 := R2 + R3? In short, every instruction will use a similar notation: a keyword for the operation, followed by the operands separated by commas. This notation is also related closely to the way instructions are represented in memory, which we'll see later.

An arithmetic instruction performs just one operation. Several instructions are needed to evaluate a larger expression. In general, you'll need a separate instruction for every operator that appears in an expression.

Example: calculate 6 + 2 * 4 and put the result into R10. We have to put the numbers into registers, using lea, and then perform the arithmetic. It doesn't matter which registers are used (as long as we avoid R0 and R15).

```
lea   R1,6[R0]    ; R1 := 6
lea   R2,2[R0]    ; R2 := 2
lea   R3,4[R0]    ; R3 := 4
mul   R2,R2,R3    ; R2 := R2*R3 = 2*8 = 8
add   R10,R1,R2   ; R10 := R1+R2 = 6+8 = 14 (hex 000e)
trap  R0,R0,R0    ; halt
```

This is nearly enough to constitute a complete program. Only one more thing is needed: a way to terminate the program when it finishes. There is a special instruction to do this: a trap instruction, where the first operand is R0, will stop the program.

```
trap  R0,R0,R0   ; halt
```

Here is a complete program named ConstArith. It consists of

1. Comments that identify the program
2. Comments that say what it does
3. The instructions
4. A trap instruction to terminate execution

```
; ConstArith: Illustrate lea and arithmetic instructions.
; Use lea to load constants into registers, then do
; arithmetic.  Sigma16:
   \url{https://jtod.github.io/home/Sigma16/}
; John O'Donnell, 2025

; Calculate 6 + 2 * 4 and put the result into R10
; Use lea to put a constant into a register
; Use mul and add to do arithmetic

    lea   R1,6[R0]    ; R1 := 6
    lea   R2,2[R0]    ; R2 := 2
    lea   R3,4[R0]    ; R3 := 4
    mul   R2,R2,R3    ; R2 := R2*R3 = 2*8 = 8
    add   R10,R1,R2   ; R10 := R1+R2 = 6+8 = 14 (hex 000e)
    trap  R0,R0,R0    ; halt
```

2.3.1 Running an example program

You can go to the Editor, and type it in (or copy and paste it in), but this program is part of the collection of examples built in to Sigma16. Here's how to run it:

- Go to the **Examples** page. Click **Examples**, then **Core**, then **Simple**. This brings up a list of small and simple example programs. Click **ConstArith.asm.txt**. You should see the listing of the program. The file name has an extension **.asm.txt** that identifies the file as an assembly language source program.
- Click **Editor**, and you should see the text of the program in the window.
- Go to the **Assembler** page and click **Assemble**.
- Go to the **Processor** page. Click **Boot**, then **Step** repeatedly and watch the effect of each instruction by observing how the registers are changed.

The Processor page shows numbers in hexadecimal. The add instruction puts 14 into R10, and this is displayed as hex 000e.

It's a good idea to step through the program slowly, rather than running it to completion at full speed. The emulator will show the next instruction to be executed, highlighted in blue. Think about what the instruction should do; in particular what changes to the registers will occur? Then click Step and check to see if the right thing happened.

Generally you can use any register you like (apart from R0 and R15), and the choices of registers in the previous examples are arbitrary. Registers R1 through R14 behave the same. However, two of the registers are different:

- **R0** contains the constant 0 and it will never change. Any time an instruction uses R0, the value it gets will be 0. It is legal for an instruction to attempt to modify R0 (for example, add R0,R3,R4 is legal) but after executing this instruction R0 still contains 0. The reason for this is that we frequently need to have access to a register containing 0.

- **R15** is used for two specific purposes. We have already seen the first: the divide instruction places the remainder into R15. The second purpose is that R15 contains the **condition code**, which is a word that contains a number of bits that provide some information about an instruction. For example, if an addition produces a result that is too large to fit in a register, a special flag indicating this is set in R15. Several of the instructions, particularly the arithmetic instructions, change the value of R15 as well as placing the result in the destination register. For this reason, R15 cannot be used to hold a variable since its value would be destroyed almost immediately.

To summarise, Registers R1 through R14 are all identical and can be used for variables. R0 contains 0 and will never change. R15 changes frequently and can be used to detect various error conditions and other information about an instruction.

Here is another example:

- Suppose we have variables a, b, c, d
- And suppose we already have these variables in registers
- Choose a register for each variable: R1=a, R2=b, R3=c, R4=d
- We wish to compute R5 = (a+b) * (c-d)

```
add   R6,R1,R2     ; R6 := a + b
sub   R7,R3,R4     ; R7 := c - d
mul   R5,R6,R7     ; R5 := (a+b) * (c-d)
```

Summary.

- A lea instruction of the form **lea Rd,const[R0]** will put the constant into Rd. It can also be written as **lea Rd,const**.
- The general form of an arithmetic instruction is **op d,a,b**. The meaning is $\mathbf{R_d := R_a\ op\ R_b}$, and the fields are:
  - op is one of add, sub, mul, div
  - Rd is the destination register (where the result goes)
  - Ra is the register containing the first operand
  - Rb is the register containing the second operand

2.4 Keeping variables in memory

So far we have used registers in the register file to hold variables. However, there are only 16 of these, and two have special purposes (R0 and R15). That leaves only 14 registers for general use, and most programs need more than 14 variables.

To solve this problem, the computer contains another subsystem called the **memory**. The memory contains a sequence of **memory locations**, each of which can hold a word (16 bits). Each location is identifed by an **address**, which is a natural number $0, 1, 2, 3, \ldots, 65535$. The notation **M[a]** denotes the value stored in the memory location with address a.

The register file and the memory serve different purposes:

- The register file is used to perform calculations. In computing something like x := (2*a + 3*b) / (x-1), all the arithmetic must be done using the register file. For example, the instruction add R1,R5,R3 has both operands and destination in registers, and this is true for all arithmetic instructions. The purpose of the registers is to provide fast access to a small number of variables while they are needed for calculations.

- There are only a few registers available, so we can't keep all the variables permanently in registers. The memory is much larger: it contains 65,536 locations so it can hold all the variables in a program. But the memory is slow, and it has a limitation: there is no instruction that can do arithmetic on data in the memory. The purpose of the memory is to hold large amounts of data for future use.

Normally, a program keeps its variables in memory, so a variable name refers to a memory location. The variable name just stands for the *address* of the location that contains the variable. This allows you to refer to a variable by a name (x, sum, count) rather than an address (003c, 0104, 00d7).

2.4.1 load and store

Since we need a lot of variables, they need to be kept in memory. But since we need to do arithmetic, and arithmetic can be performed only on data in registers, we adopt the following strategy:

- Keep variables permanently in memory.
- When you need to do arithmetic on some variables, copy them from memory to registers.
- When finished, copy the results from registers back to memory.

Two instructions are needed to do this:

- **load** copies a word from a memory location into a register. Suppose **xyz** is a variable in memory; then to copy its value into R2 we write **load R2,xyz[R0]**.
- **store** copies a word from a register into a memory location. If R3 contains the result of some calculations, and we want to put it back into memory in a variable named result, we would write **store R3,result[R0]**.

At this point we have enough instructions to write an assignment statement in assembly language. Typically we will first write an algorithm using higher level language notation, and then translate it into instructions. Translating from a high level language to assembly language is called "compiling".

Example: translate **x := a+b+c** into assembly language.

Solution:

```
load   R1,a[R0]      ; R1 := a
load   R2,b[R0]      ; R2 := b
add    R3,R1,R2      ; R3 := a+b
load   R4,c[R0]      ; R4 := c
add    R5,R3,R4      ; R5 := (a+b) + c
store  R5,x[R0]      ; x := a+b+c
```

2.4.2 The data statement

The variables used in a program need to be defined and given an initial value. This is done with the **data** statement. The variable name comes first, and it must start at the beginning of the line (no space before it). Then comes the keyword **data**, followed by the initial value, which may be written in either decimal or hexadecimal. These pieces of the statement must be separated by white space.

For example, to define variables x, y, z and give them initial values:

```
x    data   34    ; x is a variable with initial value 34
y    data    9    ; y is initially 9
z    data    0    ; z is initially 0
abc  data  \$02c6  ; specify initial value as hex
```

The data statements should come **after** all the instructions in the program. This may look surprising: in some programming languages you have to declare your variables at the beginning, before using them. There is a good reason why we will put the instructions first, and the data statements after; but the reason will come later.

Here is a simple example of a complete program that uses load, store, and data statements, to implement z := x + y.

```
; Program Add.  See README in top Sigma16 folder A minimal
; program that adds two integer variables

; Execution starts at location 0, where the first instruction
; will be placed when the program is executed.

      load   R1,x[R0]   ; R1 := x
      load   R2,y[R0]   ; R2 := y
      add    R3,R1,R2   ; R3 := x + y
      store  R3,z[R0]   ; z := x + y
      trap   R0,R0,R0   ; terminate

; Expected result: z = 37 (0025)

; Static variables are placed in memory after the program

x     data  23
y     data  14
z     data   0
```

Now you can run the program:

- Go to the Examples page. Click Core, then Simple, then Add.
- Click Editor, and you should see the text of the program in the window.
- Go to the Assembler page. Click Assemble.
- Go to the Processor page. Click Boot, then Step repeatedly and watch the effect of each instruction by observing how the registers and memory are changed.

The processor page shows the contents of the memory. For each memory location, the address is shown followed by the contents of that location. For example, if a line in the memory display shows 00b7 4c3f, this means that the memory location with address 00b7 contains 4c3f. This is written as M[00b7] = 4c3f.

Memory addresses and contents, as well as register contents, are shown in hexadecimal with no leading $. Since all these numbers are hexadecimal,

putting $ in front (or 0x as in C) just adds distraction without telling you anything useful.

The processor displays two independent views into the memory. This is convenient because it's frequently necessary to look at two different portions of the memory at the same time. For example, it's helpful to look at the machine language code in one view and the data in the other view. With programs that use advanced techniques (e.g. pointers and activation records) it's essential to be able to view two different areas of memory. Despite the two views, there is just one memory.

2.4.3 Relationship between arrays, registers, memory

The register file and the memory are similar in many ways, and they are also similar to ordinary arrays in some programming languages. The essential difference is that registers are fast but there are only a few of them, whereas memory is large but slow.

	array	reg file	memory
Type of element:	declared	word	word
Size of element:	declared	16 bits	16 bits
Element notation:	a[i]	R3	M[a]
Element is called:	element	register	location
Index is called:	index	reg number	address
Range of index:	declared	0..15	0..65535
Number of elements:	declared	**small**	**large**
Access time:	slow	**fast**	**slow**

Why does the computer have both registers and memory? After all, this makes programming a little more complicated. You have to keep track of which variables are currently in registers, and you have to use load and store instructions to copy data between the registers and memory. Wouldn't it be easier just to get rid of the distinction between registers and memory, and do all the arithmetic on memory?

Yes, this would be simpler, and there have been real computers like that. However, this approach makes the computer slower. With modern circuits, a computer without load and store instructions (where you do arithmetic on memory locations) would run approximately 100 times slower. So nearly all modern computers do arithmetic in registers, and use instructions like load and store to copy data back and forth between registers and memory.

2.5 Assembly language

The programs we have seen so far are written in **assembly language**. The machine itself executes programs in **machine language**, which is covered later. Assembly language is translated to machine language by a program called an **assembler**.

The purpose of assembly language is to give the programmer absolute control over the machine language program without having to remember lots of numeric addresses and codes. Assembly language is readable for humans, while machine language is executable by machines. For example, it is easier to remember the name "mul" for multiply than to remember the machine language code (which happens to be 3). Similarly, it's easier to remember the names of variables (x, y, sum, total) than the numeric addresses of the memory locations that hold these variables.

The syntax of assembly language is simple and rigid. Every statement must fit on one line of source code; you cannot have a statement that spans several lines, and you cannot have several statements on one line.

Sigma16 assembly language uses a small set of characters. Any character not on this list will generate an error message. A Sigma16 program can **manipulate** any 16-bit character, but the source assembly language code is restricted to this source character set. There are many characters that look similar but are actually distinct. For example, the minus sign, the hyphen, the en-dash, and the em-dash all look similar – you have to look closely to see the difference – but Sigma16 assembly language uses the minus sign, and the hyphens and dashes won't work.

These are the legal characters in an assembly language program:

- letters
  - _abcdefghijk
  - lmnopqrstuvwxyz
  - ABCDEFGHIJKLMN
  - OPQRSTUVWXYZ
- digits: 0123456789
- separators: space tab ,;
- quotes: " '

- punctuation: ".$[]()+-*
- other: ?<=>!%^&{}#~@:|/\'

Word processors often substitute characters. For example, when you type a minus sign in a paragraph of English text, word processors may replace the minus sign with a hyphen or dash, which is correct for typeset English but incorrect for assembly language. The Sigma16 editor will insert the correct characters, as will plain text editors.

Each statement has a rigid format that consists of up to four **fields**. The fields must be separated by one or more spaces, and a field cannot contain a space. Every field is optional, but if a field is missing then the following fields must also be missing, except for an optional comment. The fields are:

- label (optional) – If present, the label must begin in the first character of the line. If a line starts with a space, then there is no label field. A label has the same syntax as names or identifiers in many languages: it may contain letters, digits, underscores, and must begin with a letter. Both upper and lower case letters are allowed, and the syntax is case sensitive (Loop and LOOP and loop are three different labels).

- mnemonic – This is the name of the operation: load, lea, add, sub, etc. The mnemonic must be preceded by white space, and it must be the name of a valid instruction or assembler directive.

- operands field – the operands required by the type of statement. There are several formats possible for the operands field, depending on the instruction; these are detailed later. For example, for the add instruction the operand field must consist of three registers, separated by commas (e.g. R1,R2,R3). Spaces are not allowed in the operands field: R1,R2,R3 is fine but R1, R2, R3 is an error.

- comments – anything that follows the operands field, or anything that appears after a semicolon, is a comment. The semicolon is not required if the mnemonic and operands fields are present, but it is good practice to include it.

Here are some syntactically valid statements:

```
loop   load   R1,count[R0]    ; R1 := count
       add    R1,R1,R2        ; R1 := R1 + 1
```

Each of the following statements is wrong!

```
   add   R2, R8, R9    ; spaces in the operand field
loop1  store x[R0],R5  ; wrong order: should be R5,x[R0]
    addemup            ; invalid mnemonic
 loop2  load R1,x[R0]  ; Space before the label
```

If you forget some detail, look at one of the example programs.

When the assembler is translating a program, it begins by looking at the spaces in order to split each statement into the four fields. This happens before it looks at the operation and operands. The assembly listing uses colors to indicate the different fields. If you get a syntax error message, the first thing to check is that the fields are what you intended. For example if you meant to say

```
     add R1,R2,R3  ; x := a + b
```

but you have a spurious space, like this

```
     add R1, R2,R3  ; x := a + b
```

the assembler will decide that the mnemonic is add, the operands field is "R1," and all the rest - "R2,R3 ; x := a + b" – is a comment, and the colors of the text in the assembly listing will show this clearly.

The rules above mean that a label must begin in the first character of a line, and also that if any non-space character appears at the beginning, it is taken to be a label. The following statement has a label "add", an operation "R1,R2,R3" (and of course there is no such instruction), and the operand field is ";". Again, the syntax highlighting will show the error clearly.

```
add R1,R2,R3  ; x := a + b
```

In assembly language, you can write constants in either decimal or hexadecimal.

- decimal: 50 -39
- hexadecimal: $003b

Examples:

```
    lea   R1,40[R0]      ; R1 = 40
    lea   R2,\$ffff[R0]   ; R2 = -1

x   data  25
y   data  \$2c9e
```

There are two instruction formats in the Core architecture, which differ in the form of the operands:

- **RRR** instructions have an operand field containing three registers separated by commas. Example: **add R8,R13,R6**.
- **RX** instructions have an operand field that specifies a register and an address. The address is a name or constant, followed by a register. Examples: **load R12,array[R6]** and **lea R5,23[R0]**.

It isn't enough just to get the assembler to accept your program without error messages. Your program should be clear and easy to read. This requires good style. Good style saves time writing the program and getting it to work, while aA sloppy program looks unprofessional and makes testing and debugging harder. Here are a few tips.

Write good comments. You should include good comments in all programs, regardless of language. Comments are especially important in assembly language, because the code tends to need more explanation. At the beginning of the program, use comments to identify the program and author and to say what it does. Use full line comments to describe in general what's going on, and put a comment on every instruction to explain what it's doing.

Indent your code consistently. Each field should be lined up vertically, like this:

```
    load   R1,three[R0]  ; R1 = 3
    load   R2,x[R0]      ; R2 = x
    mul    R3,R1,R2      ; R3 = 3*x
    store  R3,y[R0]      ; y = 3*x
    trap   R0,R0,R0      ; stop the program
```

Not haphazard like this:

```
   load   R1,three[R0]     ; R1 = 3
 load  R2,x[R0] ; R2 = x
      mul R3,R1,R2           ; R3 = 3*x
store         R3,y[R0]      ; y = 3*x
  trap  R0,R0,R0      ; stop the program
```

The exact number of spaces each field is indented isn't important; what's important is to make the program neat and readable.

Spaces, not tabs. To indent your code, always use spaces – avoid tabs. In general, never use tabs except in the (rare) cases they are required (for example in makefiles). The tab character was introduced long ago into computer character sets to try to mimic the tab key on old mechanical typewriters. Unfortunately, software does not handle tab characters consistently. If you use tabs, your program can look good in one application and like a mess in another. It's easy to indent with spaces, and it works everywhere.

Some programming languages use indentation to indicate program structure. If some of that indentation is done with tabs, it could work on one computer but give syntax errors on another system. Even worse, it could compile but give different results as the program structure has changed. (Wrong results are worse than error messages.)

2.6 The Editor

To run an assembly language program, you first need to put it into the editor window. Then you can go to the Assembler page, assemble it and run it on the Processor page. There are several ways to enter a program into the editor window.

- **Starter program.** Editor tab. Click Hello World. This will load a small fixed example program into the editor buffer.

- **Collection of examples.** Examples tab. The Sigma15 app provides a collection of example programs. Click Core to see the examples that use only the core instruction set. Click Simple, then ConstArith. Go to the Editor tab and the example you selected will be in the window. All of these example programs are in the Sigma16 system, not on your local computer.

- **Type it in.** Editor tab. Type in a program. If there was already some text there, click New.

- **Read from a file.** Editor tab. Click Open File. This brings up a file chooser dialog where you can traverse the file system on your computer and select a file.

- **Copy from file and paste.** If your program is visible in another window on your computer, you can copy its text, then go to the Sigma16 editor, and paste. Normally it's better to use Open File, but if your browser has disabled file access the copy-paste method is ok.

You can edit the text in the Editor window:

- Click the arrow keys or the left mouse button to move the cursor.

- Click left mouse button and drag to select text.

- Click right mouse button and select Cut, Copy, Paste, Undo.

Save your work. After typing in a program, or editing it, you should save it to a file. Click Save As. This brings up a dialog where you can traverse your file system and provide a file name (which should end in .asm.txt).

If you leave the browser without saving the text in the editor window to a file, that text will be lost, so it's important to save your work regularly. If you don't do that, sooner or later the system will crash and you'll lose your data. When you close a browser tab that's running Sigma16, the browser may put up a message warning you that "Changes you made may not be saved". The browser can't tell whether the editor text needs to be saved.

The Sigma16 app will know what file is associated with your program if you have read it from a file (Editor: Open File) or if you have saved it to a specific location in your computer (Editor: Save As).

Once the app knows which file contains your program, you can edit it and save it again just by clicking Editor: Save. This is easier than Save As because you don't need to traverse the file system again.

You can also edit your assembly language program using an external text editor, rather than the simple editor in the Sigma16 app. If you have modified the file using an external editor, click Editor: Refresh to reload the file from your disk.

If you do use an external text editor, use a real text editor (emacs, vim, notepad++, or whatever you prefer), not a word processor. Word processors embed extra formatting data into the file, and the assembler can't handle that. Furthermore, word processors often change the characters you type. For example, they may change the minus character (-) into an en-dash. There are four different characters that look similar to a minus sign (minus, hyphen, en-dash, em-dash) and the assembly language only accepts the minus sign. If you get bad characters, the assembler will give an error message.

2.7 Jumps and conditionals

Conditionals allow a program to decide which statements to execute based on Boolean expressions. One example is the if-then statement, for example:

```
if x<y
  then statement 1
statement 2
```

A related form is the if-then-else statement:

```
if x<y
  then statement 1
  else statement 2
statement 3
```

Most high level control constructs can be translated into code that contains just one form of conditional, the **conditional goto**. This uses a Boolean expression *bexp* to decide whether to jump to *someLabel*, or not to jump:

```
if bexp then goto someLabel
```

The commonest case is where *bexp* is a comparision between two integers:

```
if x < y then goto someLabel
```

This statement is implemented in assembly language in two steps:

1. First, a **comparison** instruction is used to produce a Boolean result, which is placed in the *condition code* (which is in R15).

2. Second, a **conditional jump** instruction will either jump or not jump, depending on the condition code. This allows a choice of what instruction to execute next.

The cmp instruction compares the integers in two registers, and it sets R15 to the result of the comparison. R15 is a special register because several instructions, including cmp, use it automatically without specifying R15 in the instruction. Reflecting its special status, R15 also has a name: it's called the *condition code*.

Any relational operation can be used; it isn't limited to less-than. You can think of the condition code as holding a result like "less than", or "equal", etc. The way these comparison results are represented will be covered later.

After setting the condition code with cmp, the program executes a conditional jump. The instruction jumplt someLabel[R0] will go to the instruction with the specified label if and only if the condition code indicates "less than" (lt).

Here is a high level language statement:

```
if x < y then goto someLabel
```

To implement this in assembly language, we use a cmp instruction followed by jumplt:

```
    load   R2,x[R0]       ; R2 := x
    load   R3,y[R0]       ; R3 := y
    cmp    R2,R3          ; compare x and y
    jumplt someLabel[R0]  ; if x < y then goto someLabel
    xxx                   ; this executes if x >= y
    xxx
    xxx
someLabel
    xxx                  ; this executes if x < y
```

The conditional jump instructions have the form jumpXX, where XX is a relation, such as lt, eq, and so on:

```
jumplt  someLabel[R0]  ; if <  then goto someLabel
jumple  someLabel[R0]  ; if <= then goto someLabel
jumpeq  someLabel[R0]  ; if =  then goto someLabel
jumpne  someLabel[R0]  ; if != then goto someLabel
jumpge  someLabel[R0]  ; if >= then goto someLabel
jumpgt  someLabel[R0]  ; if >  then goto someLabel
```

These conditional jumps treat the contents of the registers as integers represented in two's complement notation. This means, for example, that $ffff is less than 0, because $ffff represents -1. There are several more conditional jumps that you can use for comparing natural numbers (binary), and a few other things as well.

A compare instruction is often followed immediately by a conditional jump, as in the example above. You can also save the result of a comparison in a Boolean variable and perform logic on Boolean variables. The Boolean variables can later be used to control conditional jumps. This topic will be discussed later, as it involves instructions from the full standard instruction set.

The address in a jump instruction – the place to jump to – is normally specified as a label which is defined in the label field of some instruction. You can place a label in the same line as the instruction, or it can be on a line with nothing else, in which case the label refers to the next instruction. In the following code, label1 is the address of the add instruction and label 2 is the address of the sub instruction.

```
label1   add  R2,R4,R13
label2
         sub  R15,R0,R1
```

If-then constructs are translated into assembly language following two similar fixed patterns. Suppose Bexp is a Boolean in any register Rd

```
if bexp
  then statement 1
statement 2
```

This is translated according to the following pattern:

```
     if !bexp then goto L2
     statement 1
L2:
     statement 2
```

Here is an example:

```
a := 93
x := 35
y := 71
if y > x then a := 59
b := 104
```

The corresponding assembly language is:

```
; a := 93
      lea     R1,93[R0]     ; R1 := 93
      store   R1,a[R0]      ; a := 93

; x := 35
      lea     R1,35[R0]     ; R1 := 35
      store   R1,x[R0]      ; x := 35

; y := 71
      lea     R1,71[R0]     ; R1 := 71
      store   R1,x[R0]      ; x := 71

; if y > x
      load    R1,y[R0]      ; R1 := y
      load    R2,x[R0]      ; R2 := x
      cmp     R1,R2         ; compare y with x
      jumple  R3,skip[R0]   ; if not y > x then goto skip

;  then a := 59
      lea    R1,59[R0]      ; R1 := 59
      store  R1,a[R0]       ; a := 59

; b := 104
skip  lea    R1,104[R0]     ; R1 := 104
      store  R1,b[R0]       ; b := 104
```

Notice the use of jumple: if the Boolean expression (y>x) is False we want to skip over the "then" part, so we want to jump if y <= x (hence jumple).

An if-then-else statement has a similar compilation pattern, but this time there are two separate parts: the "then-part" and the "else-part". Depending on the value of the Boolean expression, one of those parts should be executed and the other should be skipped over.

For if-then-else, and many other control constructs, we need an **unconditional jump** which will always go to the specified address, and which doesn't use a Boolean.

```
jump   somewhere[R0]    ; go to somewhere
```

The general form of an if-then-else is

```
if x < y
  then S1
  else S2
S3
```

The general if-then-else construct can be translated to use just goto and conditional goto:

```
    if x >= y then goto L2
    S1
    goto L3
L2: S2
L3: S3
```

2.8 Loops

Loops are implemented using compilation patterns based on comparisons and jumps. The fundamental form is the **while loop**.

```
while Bexp do S1
S2
```

The compilation pattern is:

```
L1   if not Bexp then goto L2
     S2
     goto L1
L2
```

Occasionally you may encounter an infinite loop, which is sometimes expressed as a while loop:

```
while true do S1
```

This doesn't need a Boolean expression; it is simply compiled into:

```
loop
   instructions for S1
   jump   loop[R0]
```

Infinite loops are rather rare, or at least they should be. On occasion they are exactly what is wanted. For example, operating systems contain a loop that looks for something useful to do, and then does it, and this should be an infinite loop.

However, there is a common but poor programming style that uses infinite loops with random break or goto statements to get out of the loop. This may be appropriate on occasion but generally it is bad style.

So far we have seen several compilation patterns:

- if-then
- if-then-else
- while

Every high level programming construct has a compilation pattern, and they are mostly built using comparisons and jumps. In principle, these patterns are straightforward to use. However, there are two issues that require a little care: uniqueness of labels and nested statements.

Labels must be unique: the same one cannot be used twice in the same program, and if it is, the assembler will give an error message. This means that you cannot follow the compilation patterns blindly. If you use "loop" as the label for a while loop, as in the pattern above, you need a different label for your next while loop.

The best approach here is not to use labels like loop, loop1, loop2. It's far better to think about the **purpose** of the construct in your program and to use a label that reflects this purpose.

Another complication is that most programs contain **nested statements**. These are statements that contain smaller statements, and the containment may go several levels deep.

```
if b1
  then S1
       if b2 then S2 else S3
       S4
  else S5;
       while b3 do S6
S7
```

There is an important principle to follow here: every time a statement appears in a compilation pattern (we have been calling them S1, S2, S3, etc.), it should be translated as a **block**.

A block is a sequence of instructions which **always** begins execution at the first instruction, and **always** finishes at the end. You **never** jump into the middle of it, and it **never** jumps out of the middle to some other place.

Every statement should be compiled into a block of code. This block may contain internal structure — it may contain several smaller blocks — but to execute it you should always begin at the beginning and it should always finish at the end.

In programming language theory, programming with blocks is often considered to be good practice or good style. But it is more than just an issue of style. If you always treat the statements inside compilation patterns as blocks, the patterns will "just work", no matter how deeply nested they are. If you violate the block structure, you will find it difficult to get the program to work.

2.9 Machine language

The bits representing an instruction (written in hex) (e.g 0d69) are **machine language**. The hardware runs the machine language — it's just looking at the numbers. The text notation with names – e.g. add R13,R6,R9 – is called **assembly language**. Assembly language is for humans to read and write; machine language is for machines to execute. Both languages specify the program in complete detail, down to the last bit.

As a program is running, the memory contains all your program's data: the variables, data structures, arrays, lists, etc. **The memory also contains the machine language program itself.** The program is stored inside the computer's main memory, along with the data. This concept is

called **the stored program computer**.

There is an alternative approach: a computer can be designed to have one memory to hold the data, and a completely separate memory to hold the program. This approach is often used for special-purpose computers (primarily micro-controllers), but experience has shown this to be inferior for general purpose computers.

Sigma16 has several different kinds of instruction. These are called *instruction formats*. All the instructions with the same format have similar representations in machine language. The Sigma16 Core has two instruction formats:

- RRR instructions use the registers
- RX instructions use the memory

The machine language program is in the memory. Therefore we need to represent each instruction as a word that can be stored in memory. An instruction format is a systematic way to represent an instruction using one or more words (a word is a string of bits).

- An RRR instruction is represented in one word
- An RX instruction is represented in two words.

2.9.1 RRR instructions

Every RRR instruction is represented in one word (16 bits), which is organised as four 4-bit fields. When describing a machine word, we normally write each field value as a hexadecimal digit: one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. These correspond to 0, 1, ..., 15 in decimal.

The machine language code needs to specify which RRR instruction this is. Is it add? sub? mul? another? This is done with an operation code — a number that says what the operation is. There are about a dozen RRR instructions, so a 4-bit operation code suffices. For example, the add instruction has an opcode of 0.

We also need to specify three registers: destination and two source operands. There are 16 registers, so a specific one can be specified by 4 bits. Thus the requirements for an RRR instruction are 4 fields, each containing 4 bits, for a total of 16 bits, so an RRR instruction exactly fills one word.

The four fields in an RRR instruction are named:

- op – Operation code (e.g. add, sub, etc.) This is a number that specifies which instruction it is.
- d – Destination register
- a – First source operand register
- b – Second source operand register

Here are the operation codes for the RRR instructions we have seen so far. There are a few more RRR instructions, but they all have the same form as these do.

mnemonic	opcode
add	0
sub	1
mul	2
div	3
trap	b

Don't memorise this table: you just need to understand how it's used. For example, let's translate this assembly language statement into machine language:

```
mul  R14,R6,R10
```

1. mul is an RRR instruction, so we need to find the values of the four fields: op, d, a, b.
2. The table above says that the opcode for mul is $2 (that's 0010 as a bit string).
3. The destination is R14, so the d field is $e.
4. The first operand is R6, so the a field is $6.
5. The last operand is R10, so the b field is $a.
6. Putting the fields together ("assembling them into an instruction"), we get the machine language instruction: $2e6a.

Every RRR instruction is assembled the same way. The numbers in the fields will be different, but the procedure is always exactly the same.

2.9.2 RX instructions

We have seen several RX instructions so far:

```
lea    R5,19[R0]   ; R5 = 19
load   R1,x[R0]    ; R1 = x
store  R3,z[R0]    ; z = R3
```

An RX instruction contains two operands, separated by a comma.

- The first operand is a register, and it's called the "register operand" (e.g. R5)
- The second operand is a constant or label followed by a register (e.g. 19[R0] or x[R0]). This is called the "address operand", and it contains two parts:
  - The first part is either a constant like 19, or a label like x. This part is called the "displacement", and its value is one word. If this part is a constant (e.g. 19) then the value of the displacement is simply the value of the constant. If it's a label like x, the value is the address in memory of x.
  - The seocnd part is a register in square brackets, e.g. [R0]. This is called the "index register". In the examples we've seen so far, the index register has always been R0, but in general it can be any register, and this is useful for several purposes we'll see later.

For RRR instructions, the op field contains a number saying which RRR instruction it is. This is the primary opcode. For RX instructions, the op field always contains f. So how does the machine know which RX instruction it is? This is specified by a secondary code in the b field. Here are the full codes (op and b) for a few of the RX instructions:

mnemonic	op	b
lea	$f	$0
load	$f	$1
store	$f	$2

Consider this assembly language statement:

```
lea   R2,38[R7]
```

There are two words in the machine language code. The first word has the same 4 fields as for RRR instructions: op, d, a, b, where

- op contains $f for every RX instruction
- d contains the register operand (in the example, $2)
- a contains the index register (in the example, $7)
- b contains a code indicating which RX instruction this is (1 means load). This "secondary opcode" is required because the op field just indicates that the instruction is RX format, but doesn't say which RX instruction it is.

The second word contains the **displacement**. In the example, this is 38 (or $0026).

If the address field of an instruction has a label (like x) rather than a constant (lik 204), we need to know the value of the label in order to assemble the instruction. If x is a variable, the value of the label x is the address of the variable in memory. If x is a label on an instruction, its value is the address of the instruction.

Example: translate load R7,x[R12] from assembly language to machine language, given that the address of x is $0c3a.

- First word:
  - op = $f (indicates that this is an RX instruction, but not which one; that needs the b field)
  - d = $7 (the register operand)
  - a = $c (part of the address operand: the register [R12])
  - b = $1 (this is the secondary opcode, which indicates that this RX instruction is load)
  - The first word is $f701
- Second word:
  - The displacement, part of the address operand, is $0c3a

- The complete machine language instruction: $f7c1 $0c3a

The instruction jump loop[R0] has only one operand in assembly language, which is loop[R0]. This looks like an exception: the other RX instructions have two operands. However, the machine language representation of jump also contains a register operand, just like all RX instructions, but the machine ignores that register when the program runs, and the assembler just sets it to R0 in the machine code. This is called a "don't care" operand.

2.9.3 Pseudoinstructions

We have seen conditional jump instructions like **jumplt loop**. Technically, jumplt, jumpeq and the rest are called "pseudoinstructions". They are just a convenient assembly language notations to describe the actual underlying machine language instructions. All conditional jumps are expressed in machine language using just two real instructions: jumpc0 and jumpc1:

```
jumpc0 Rd,disp[Ra]
jumpc1 Rd,disp[Ra]
```

The details of how jumpc0 and jumpc1 work will be discussed in the section on the Standard architecture. Here is a list of the pseudoinstructions for conditional jumps after an integer comparison:

```
jumplt  someLabel[R0]  ; if <  then goto someLabel
jumple  someLabel[R0]  ; if <= then goto someLabel
jumpeq  someLabel[R0]  ; if =  then goto someLabel
jumpne  someLabel[R0]  ; if != then goto someLabel
jumpge  someLabel[R0]  ; if >= then goto someLabel
jumpgt  someLabel[R0]  ; if >  then goto someLabel
```

2.10 A strange program

Consider "Program Strange" below. You can find the program on the Examples page (go to the Core/Instructions section), or you can copy it from below and paste it into the Editor page.

This program doesn't compute anything particularly useful. It's rather strange and not a model for good programming style, but it illustrates an extremely important concept, which is discussed in the solution below.

The program has a variable y which is defined in the last line: y data 0. But you can edit that line to give y any initial value; it doesn't need to be 0.

The aim of this tutorial is to figure out what the program does, and to do so for several specific values of y. A solution is given below, but please try first to work it out for yourself.

2.10.1 The problem

Run the program with several different initial values of the variable y. These values are given below. For each run, assume that all the registers contain 0 after the program is booted, before it begins execution. For each value of y, do the following:

1. Edit the last statement of the program to give y the specified value.
2. Execute the program manually, with paper and pencil but without the computer. Give the final values of the registers. Think about what is going on as the program runs.
3. Assemble the program and boot it in the processor. Run the program on the emulator and check whether your execution was correct. Be sure to single-step through the program. Before executing an instruction, think about what it will do. Then click Step and look at the registers to see if you were right.

Here are the different variations:

1. Run the program in its original form, with **y data 0**
2. Change the last line to **y data 1** and run it again
3. Now use **y data 256**
4. **y data 8192**
5. **y data -5424**

And here is the program:

```
; Strange: A Sigma16 program that is a bit strange
        load   R1,y[R0]
        load   R2,x[R0]
        add    R2,R2,R1
        store  R2,x[R0]
        lea    R3,3[R0]
        lea    R4,4[R0]
x       add    R5,R3,R3
        add    R0,R0,R7
        trap   R0,R0,R0
y       data   0
```

2.10.2 Solution

It's best to try answering the questions on your own first, and then to check by running the program on the emulator, before reading the solution!

The program loads an **instruction** into a register, does arithmetic on it by adding **y** to it, and stores the result back into memory. This phenomenon is called **self-modifying code**, and it exploits the fact that instructions and data are held in the same memory (this is the **stored program computer** concept). The original instruction is **add R5,R3,R3**, and its machine language code is 0533.

1. When y=0, the final values are: R1=0, R2=0533, R3=3, R4=4, R5=6. The only notable points are that the store instruction doesn't actually change the value of the word in memory (it was 0533 and 0533 is being stored there), and the last add instruction doesn't change the value in R0 because R0 can never change; it is always 0. (Of course if R7=0 then the result of the addition is 0 anyway.)

2. When y=1, the final values are: R1=1, R2=0534, R3=3, R4=4, R5=7. Note that R5 is **not** 3+3=6. When y=1 is added to the instruction, the result is 0534 which means **add R5,R3,R4**, so instead of adding R3+R3 it adds R3+R4.

3. When y=256, the final values are: R1=256=0100, R2=0633, R3=3, R4=4, R5=0, R6=6. The decimal number 256 is 0100 in hexadecimal.

When this is added to the instruction, the result is 0633, which means **add R6,R3,R3** so R3+R3 is loaded into R6, not into R5.

4. When y=8192, the final values are: R1=4096=2000, R2=2533, R3=3, R4=4, R5=9. The decimal number 8192 is 2000 in hexadecimal, and when this is added to the instruction the result is 2533, which means **mul R5,R3,R3**. It's no longer an **add** instruction, it's a **multiply** instruction that calculates R5 := R3*R3 = 9.

5. When y=-5424 the program goes into an infinite loop. R1=ead0 (the hexadecimal representation of -5424), R2=f003, R3=3, and R4=4. What started out as the **add** instruction at x has been transformed into **jump 7[R0]**, comprising the word at **x** (f003) and the following word (which is 0007). This jump instruction goes back to the first lea instruction, and the program runs for ever (lea, lea, jump).

There is a lot to say about the phenomenon of self-modifying code.

This program shows clearly that a computer does not execute assembly language; it executes machine language. Try running it on the Sigma16 application (single step each instruction). You'll see that the assembly language statement **add R5,R3,R3** is highlighted in red, but that is just the GUI trying to be helpful. What's important is that the machine language instruction is fetched from memory and loaded into ir (the instruction register), and that is not 0533. The machine decodes the contents of ir and does whatever that says to do; it isn't aware of the assembly language statement. Indeed, a machine doesn't even understand the concept of assembly language — everything is just bits!

To follow exactly what is happening in the emulator, it's important to look at the pc and ir registers. These reflect what the machine is doing. The assembly language does not.

What is self-modifying code good for? The answer lies in the early history of electronic computers. Early computers (late 1940s and early 1950s) did not use an effective address (i.e. displacement + index) like Sigma16; the instructions simply specified the absolute memory address of an operand. This is ok for simple variables, but how could they process arrays?

The solution was to use self modifying code. In a loop that traverses an array, there would be a load instruction using address 0. In the body of the loop, there would be instructions to calculate the address of x[i] by loading the address of x and adding i; this is then stored into the address field of the

load instruction. That instruction is then executed, obtaining the value of x[i]. This technique became obsolete in the early 1950s with the invention of index registers and effective addresses.

The pioneers of computers considered the concept of the **stored program computer** (i.e. the program and data are in the same memory) to be fundamental and essential. One of the most important reasons was that it made arrays possible. Now we consider the stored program concept to be fundamental **for different reasons**.

Self modifying code is tricky, and difficult to debug. It makes programs hard to read: you can't rely on what the program says, but on what its instructions will become in the future. For these reasons, self modifying code is now considered to be bad programming practice.

If a program modifies itself, you can't have one copy of the program in memory and allow it to be shared by several users. For example, it's common now to have a web browser open with several tabs. Each tab is served by an independent process (a separate running instance of a program that updates the window showing the web page). If you have 5 tabs open, there are 5 processes, each running the same machine language code, and there's only one copy of that in memory. This wouldn't work if the program modified itself!

Self modifying code leads to security holes: if a hacker has the ability to change your machine language code in memory, they could make your own program act against you.

Modern computers use a technique called **segmentation** that prevents a program from modifying itself. This leads to increased reliability and security.

Some computers have a facility that allows you to gain the power of self modifying code without actually modifying the code in memory. The idea is to have an instruction **execute R1,x[R0]** which calculates the logical or of the two operands and then executes the result; x is the address of an instruction and R1 contains the modification to it. The modified instruction is executed, but there is no change to the machine code in memory. This idea was used in the IBM 360 and its successors. However, as the design of effective addresses has become more sophisticated, the execute instruction is rarely needed, and most modern computers don't provide it.

2.11 Breakpoints

When you are testing or debugging a program, you may need to execute many instructions before reaching the point you're interested in. Some programs execute thousands of instructions just to initialize. It's infeasible to step through all those instructions, yet if you just run at full speed you won't be able to see what's happening in the section you are working on.

The solution is to run the program at full speed but to force it to stop when it reaches a specific instruction. This is called a *breakpoint*. When the program stops at the breakpoint, you can examine the registers and memory, and step through instructions from that point. At any time you can click Run and full speed execution resumes, until either another breakpoint is encountered or the program terminates.

Both a breakpoint and a halt instruction (trap R0,R0,R0) will stop execution of the program. The difference is that after a breakpoint you can click Step or Run to continue, but after halt the program cannot execute any more instructions until you boot the processor again.

There are two ways to set a breakpoint:

- *Trap break*: Insert an instruction into the program that breaks execution at that point.
- *External break*: Define a breakpoint in the user interface, without modifying the program.

Both forms are useful. Most of the time, while debugging a program, a trap break is easier and more convenient. However, if you don't want to modify the program or reassemble it, or if you realise that you need a breakpoint after execution has already started, then an external break is better.

2.11.1 Trap break

A trap break is a trap instruction whose first operand register contains the value 4. The other operand registers are ignored. When this instruction is executed, the emulator will stop execution, and you can resume execution later by clicking Step or Run.

Suppose you want to check what the load instruction is doing in this code:

```
\ldots{}
add    R1,R2,R3
load   R4,x[R1]
\ldots{}
```

Insert a breakpoint just *before* the instruction you want to examine. The breakpoint requires two instructions. The first instruction loads the break code into some register (say R9 but it doesn't matter which), and the second instruction is a trap which performs the break. The first operand is the register that contains the break code, and the other two operands are ignored, so we can just use R0.

```
\ldots{}
add    R1,R2,R3
lea    R9,4       ; R9 := trap break code
trap   R9,R0,R0   ; breakpoint
load   R4,x[R1]
\ldots{}
```

Now you can run the program at full speed, but when it executes the trap instruction, the emulator will stop. Since the trap instruction has just executed, it will be highlighted in red, and the instruction you're interested in – the load – will be highlighted in blue. You can single step for a while, and click Run again at any time to resume full speed execution.

A common technique is to put a trap break at the beginning of a loop. By clicking Run repeatedly, you can step through the loop iterations.

For an example of a long running program with a trap break, see Examples / Core / Testing / Looper.

2.11.2 External break

An external break tells the emulator to perform a breakpoint without modifying the program. Use these steps to set an external break:

1. Find the address of the instruction to stop at: look at the assembly listing, find the instruction, and the listing gives its address.

2. Go to the processor page, click Boot and then click Breakpoint.

3. A small window will appear; type in the breakpoint address. It must be a hexadecimal address in assembly language format: it must begin with a $ and then contain four hex digits. No other characters may be present, not even white space.

4. Click Refresh. This parses the address you entered and remembers it. (If you change the address in the window, click Refresh again.)

5. Click Enable. This turns on the breakpoint.

6. Click CLose. The breakpoint popup window will disappear so you can see the Processor again.

Now click Run and the program will execute at full speed. When the pc register is equal to the breakpoint address, the emulator will stop. Then you can Step or Run to continue execution.

As long as the breakpoint is enabled, execution will stop every time that location is encountered. To prevent this, open the breakpoint popup again and click Disable.

3 Standard architecture tutorials

The standard architecture provides additional instructions that simplify many programming tasks, such as logic, functions, and procedures. It also provides additional registers that control system functions, such as interrupts. The standard architecture supports systems programming, including emulators, interpreters, and operating system kernels. There are assembler directives that simplify large scale programming, modules, linking, and relocation.

Some of the instructions in the standard architecture are under development and may be changed in future versions.

3.1 Defining symbols with equ

Sometimes a program needs to use a constant. For example, you may need to use an unusual logic function that doesn't have a built-in pseudoinstruction. There are pseudoinstructions for the most common logic operations:

```
andb   R2,R7,9,13      ; R2.9 := R2.9 and R7.13
```

But suppose you want to use a less common logic function (for example, nand) that doesn't have a pseudoinstruction. You can use the general logic instruction and specify the encoding of the logic function you need. The encoding for the nand function is 14, so you need this instruction:

```
logicb R2,R7,9,13,14   ; R2.9 := R2.9 nand R7.13
```

However, it's poor programming style to use magic constants like 14 in your program. It's easy to get them wrong, it makes the code harder to understand, someone else reading the program will wonder what the 14 means, and you might even forget it yourself.

The assembler provides the **equ** statement for defining symbols. The symbol you're defining goes in the label field. The operation is equ, and the operand is the value of the symbol. Thus you can write "nand equ 14" to define the symbol:

```
nand equ     14             ; logic code for nand
```

Now you can write nand instead of 14:

```
logicb R2,R7,9,13,nand  ; R2.9 := R2.9 nand R7.13
```

An equ statement must appear before any use of the symbol that it defines.

This kind of statement, which doesn't generate an instruction but merely gives the assembler some additional information, is called a directive. It is not a machine instruction.

In a program that does lots of logic, you could define the logic functions near the beginning of the program:

```
; Symbolic names for common logic function codes
invx equ     12
and  equ     1
or   equ     7
xor  equ     6
nand equ     14
nor  equ     8
xnor equ     9
```

There are many applications for the equ statement, besides defining logic function constants. You can give a symbolic name to condition code flags you need, and for flags in system control registers, and for the indices of fields in your application program. If you have size constants, such as the number of elements in an array or the size of a stack or heap, you can define them with equ.

If you want to load the value of an equ symbol into a register, use lea:

```
lea    R3,nand[R0]  ; R3 := code for nand logic function
```

3.2 Some notations for convenience

There are several shortcuts that can be useful for experienced programmers, especially when working on large programs. But when you're starting out, it's better not to use these techniques.

3.2.1 ; before comment is optional if operand field exists

It isn't required to have a semicolon ; before every comment. If an assembly language statement has an operation field and an operand field, then everything after the operand field is taken to be a comment, so the semicolon is not required.

```
      lea   R3,42[R0]    ; This is a comment
      load  R5,xyz[R0]   and this is also a comment
```

Although most assembly language statements have an operand field, some do not. In these cases, the semicolon is required to indicate a comment.

It's good style to use a semicolon consistently to indicate a comment, even where it wouldn't be required.

3.2.2 [R0] is assumed if it's omitted

In an RX instruction, if you omit the [Reg] after the displacement, the assembler will automatically use [R0]. Thus lea R3,42 is an abbreviation for the complete form, lea R3,42[R0]. The following two assembly language statements describe the same instruction (their machine language code is identical):

```
lea  R3,42[R0]   ; R3 := 42
lea  R3,42       ; R3 := 42
```

You can write the instruction either way; both are translated to exactly the same machine language, and they execute exactly the same way. These are also equivalent:

```
jumplt loop[R0]  ; if < then goto loop
jumplt loop      ; if < then goto loop
```

3.3 Common logic operations

The Sigma16 architecture provides two general logic instructions: logicf operates on fields within a word, and logicb operates on individual bits. Those two general instructions implement all possible logic functions on two operands, not just inv, and, or, xor. See the Architecture section below for an explanation of logicf and logicb.

To simplify programming, Sigma16 also provides pseudoinstructions for commonly used logic operations on words and bits. These are described below, and they are sufficient for many applications. For each pseudoinstruction, the assembler generates an equivalent logicf or logicb instruction.

3.3.1 Word logic

Word logic operations are "bitwise": bit i of the result depends on bit i of each operand, and does not depend on bits at any other index. There are

pseudoinstructions for the most common word logic operations: invw, andw, orw, xorw.

The invw pseudoinstruction inverts all the bits in a register:

```
lea   R3,\$f0f0[R0]   ; R3 := f0f0
invw  R3             ; R3 := 0f0f
```

The other word logic pseudoinstructions take two register operands. They calculate the result and place the result in the first operand register, overwriting its previous value. There is not a separate destination register, as in the add instruction.

For example, andw R1,R2 calculates the logical *and* of R1 and R2, and places the result in the destination register R1.

```
lea   R1,\$f0f0[R0]   ; R1 := f0f0
lea   R2,\$00ff[R0]   ; R2 := 00ff
andw  R1,R2          ; R1 := 00f0
```

The orw (logical inclusive or) and xorw (logical exclusive or) instructions are similar.

```
lea   R1,\$f0f0[R0]   ; R1 := f0f0
lea   R2,\$00ff[R0]   ; R2 := 00ff
orw   R1,R2          ; R1 := f0ff
xorw  R1,R2          ; R1 := f000
```

3.3.2 Bit logic

There are also pseudoinstructions that perform logic on two individual bits, which may be at any index in any register. The result overwrites the first operand. For example, andb R1,R2,3,9 calculates R1.3 and R2.9 and stores the result in R1.3. Here are some examples:

```
   lea     R1,\$00ff[R0]   ; R1 := 00ff
   lea     R2,\$00f0[R0]   ; R2 := 00f0
   andb  R1,R2,3,9        ; R1 := 00f7
;       (R1.3 := R1.3 and R2.9 = 1 and 0 = 0)
   orb  R1,R2,11,5        ; R1 := 08f7
;       (R1.11 := R1.11 or R2.5 = 0 or 1 = 1)
   xorb  R1,R2,4,6        ; R1 := 08e7
;       (R1.4 := R1.4 xor R2.6 = 1 xor 1 = 0)
```

3.4 Bit manipulation

There are several instructions that treat a word as just a string of bits, not as a number. Some of these operate on individual bits in a word, some operate on fields within a word, and some operate on the entire word.

3.4.1 Accessing individual bits

You can set an individual bit in a register at a specified index to 0 using clearb. The first operand is a register, and the second operand is the index of the bit. All other bits in the register remain unchanged. So clearb R3,2 makes R3.2 zero (that is, the bit at index 2 in R3). All the other bits in R3 remain unchanged.

```
lea     R3,\$ffff[R0]  ; R3 := ffff
clearb  R3,2          ; R3 := fffb  (R3.2 := 0)
```

The setb pseudoinstruction is similar, but it makes the specified bit 1.

```
add     R4,R0,R0      ; R4 := 0000
setb    R4,6          ; R4 := 0040  (R4.6 := 1)
```

3.4.2 Shifting

There are two shift instructions, which shift the word in Re by f bits (to left or right) and place the result into Rd. These are logical shifts: the bits that are shifted out are discarded, and bits shifted into the word are always 0.

```
shiftl Rd,Re,f  ; Rd := Re shifted left by f bits
shiftr Rd,Re,f  ; Rd := Re shifted right by f bits
```

Here are some examples; the comments show the expected result in both hexadecimal and as bits.

```
lea    R1,2[R0]  ; R2 = 0002   0000 0000 0000 0010
shiftl R5,R1,4   ; R5 = 0020   0000 0000 0010 0000
shiftl R6,R1,13  ; R6 = 4000   0100 0000 0000 0000
shiftr R7,R6,3   ; R7 = 0800   0000 1000 0000 0000
shiftr R7,R7,11  ; R7 = 0001   0000 0000 0000 0001
```

3.4.3 Extracting fields

A common situation, especially in systems programming, is that you have a word that contains several fields. A machine language instruction is like that; also a network packet contains a number of fields. The sizes of the fields, and their positions within a word, can be anything. Software, especially systems programming, often needs to be able to extract a field from a word and put it somwhere else so it can be used.

It's possible to obtain any field from a register, and put it somewhere in another register, using a sequence of shift and logic instructions (particularly shiftl, shiftr, andw, orw). However, these sequences of instructions are error prone, hard to read, and they execute frequently in the inner loop of many systems programs.

Sigma16 provides an instruction *extract* which solves this problem with just one instruction. It allows you to extract an arbitrary field from a register and put it into an arbitrary location in another register.

The extract instruction takes operands that specify the source register (which contains the field you're interested in), the destination register (where it will put the field), the location of the field in the source register (specifed as the index of the rightmost and leftmost bits in the field), and the location where you want to put the field in the destination register.

This instruction copies 4 bits from R1.4:7 to R2:8:11:

```
extract  R2,R1,8,4,7   ; R2.8:11 := R1.4:7
```

In this example, the operands are:

- R2 is the destination register
- R1 is the source register
- 8 is the index of the rightmost bit in the destination where the field will be placed at ascending bit indices
- 4 is the index of the rightmost bit of the field in the source register
- 7 is the index of the leftmost bit of the field in the source register

This instruction is equivalent to the following (although it is just one instruction, not four instructions):

```
R2.8  :=  R1.4
R2.9  :=  R1.5
R2.10 :=  R1.6
R2.11 :=  R1.7
```

The extract instruction modifies only the specified bits in the destination register; the other bits remain unchanged. Here is an example:

```
lea      R1,\$00f0[R0]  ; R1 = 00f0
add      R2,R0,R0      ; R2 = 0000
extract  R2,R1,8,4,7   ; R2 = 0f00  R2.8:11 := R1.4:7
```

3.5 Saving and restoring contiguous registers

There are several situations where a program needs to store a sequence of registers into consecutive memory locations. Later the registers must be reloaded from the block of memory. One example is a procedure call; another example is gaining access to all the fields of a record. These operations are supported by the save and restore instructions.

The following save instruction stores R3, R4, ..., R9 into consecutive memory locations, starting at 5[R14]:

```
    save    R3,R9,5[R14]   ; save R3 to R9
```

The save is equivalent to a sequence of individual store instructions:

```
store   R3,5[R14]
store   R4,6[R14]
store   R5,7[R14]
store   R6,8[R14]
store   R7,9[R14]
store   R8,10[R14]
store   R9,11[R14]
```

To get the values from memory back into the registers, use this restore instruction:

```
 restore R3,R9,5[R14]  ; restore R3 to R9
```

The restore is equivalent to a sequence of individual load instructions:

```
load    R3,5[R14]
load    R4,6[R14]
load    R5,7[R14]
load    R6,8[R14]
load    R7,9[R14]
load    R8,10[R14]
load    R9,11[R14]
```

Using save and restore makes the code shorter and more readable, and it's easier to check that the save an restore match each other.

3.6 Modules and linking

A **module** is a section of a program; it may be the complete program or just a part of it. A module may be saved in a file or it may simply be text in the editor buffer. A program may consist of just one module, or it can be split between several files.

The Editor page contains a text area called the **editor buffer**. When you launch Sigma16, there is one module whose text is empty and displayed in the editor buffer. You can type a program (to be precise, a module) into the

editor buffer. When you switch to the Assembler page, the Assemble button will translate the text in the editor buffer to machine language, which you can execute on the Processor page.

After entering a program in the editor buffer, you should save it to a file. Click Save and the text in the editor buffer will be written to a file on your computer. Depending on how the system is configured, there may be a dialogue box asking you for a file name, or a generic default file name may be used (for example, "S16DownloadFile (2).txt" or something similar).

Another approach is to use a separate text editor, and to copy/paste text between the external editor and the Editor page on Sigma16.

To create a new module without destroying the existing one, click **New** in the editor page. This will make a new module with empty text and display that in the editor buffer, so any text you had there will disappear. However, that text isn't lost, it's just hidden, and to get it back you just need to select the previous module.

The **Modules** page shows a list of all the modules and allows you to select one to work on. The modules are shown in small sections separated by horizontal lines. The modules are numbered starting from 0, so if there are n modules their numbers go from 0 to n-1. For each module, the module number is shown, followed by some buttons to operate on that module, and some information about it. The first few lines of the module are shown. If you follow good programming style, where the first rew lines of each module identify the program, you'll be able to see at a glance what each module is without visiting it in the editor.

Several buttons appear for each module in the list. At any time, one of the modules is **selected**. Click the Select button for any module to select that one. The selected module number is highlighted in red, and when you go to the Editor page the text of the selected module appears in the editor buffer. This means you can have several programs open at the same time, and just switch from one to the other using the Select buttons in the Modules page.

You can also get rid of a module by clicking its Close button. This will delete its text, so it may be a good idea to select it and download it in the Editor before closing it.

So far we have just created new modules by clicking **New** (in either the Editor page or the Modules page). You can also read files on your computer into Sigma16. Click **Choose files** and a dialogue box will pop up. You can select one or more files, and these will now appear in the list of modules.

If a module was created by reading it from a file, its entry in the list contains an extra **Refresh** button. Clicking this will reread the file and you won't need to use the file chooser dialogue box again.

4 The Sigma16 architecture

Sigma16 contains a set of registers, an arithmetic logic unit (ALU) and functional units for arithmetic calculations, a memory, a memory management unit that provides virtual memory, an interrupt system, and an Input/Output controller using direct memory access (DMA).

4.1 Implementations

The Sigma16 software application contains a complete programming environment, using emulation to implement the processor. The programming environment includes a file manager, editor, assembler, linker, and emulator, and it provides Input/Output and secondary storage. The software runs in a web browser, and does not require any installation. There are also command line tools which can be installed on a local computer, but they are not necessary.

There is also a digital circuit, specified in a functional hardware description language, which implements the Core architecture. The circuit specification is executable, and Core programs can be executed by simulating the circuit. The circuit is suitable for implementation using either an FPGA or a custom VLSI design. In simulation, the simulation driver provides Input/Output, and in a hardware realisation these would be provided by I/O hardware.

4.2 Subsystems

A **register** is a digital circuit that can retain one word of data. A new value can be loaded into a register, and the current contents may be read out. Registers are fast, and most computation is performed using the registers. Sigma16 contains several groups of registers; each group is displayed in a box on the Processor tab.

- The **Register File** is an array of 16 registers named R0, R1, ..., R15. These registers are accessible to the machine language program.

Programs use the register file to hold variables that are currently in use.

- The **Control** registers (pc, ir, adr) keep track of the instruction that is currently executing.
- The **System** registers control the system status and interrupts.
- The **Virtual Memory** registers are used for memory management.

The **memory** is an array of 2^{16} words. Each word in the memory is identified by an **address**, which is a 16-bit natural number. The memory is similar to the register file, but significantly slower and much larger.

Computational units. The ALU (arithmetic and logic unit) is a circuit that can do arithmetic, such as addition, subtraction, comparison, and some other operations More complex operations, such as multiplication and division, are provided by functional units.

The **Input/Output** system can transfer data between the computer and the outside world.

4.3 Words

In Sigma16, a **word** is a sequence of 16 bits. Occasionally we will also refer to a **double word** (a sequence of 32 bits). A a **generic word** is a sequence of bits of arbitrary length. The system does not use bytes (a byte is 8 bits) or extended words (64 bits).

The hardware components in Sigma16 are mostly 16 bits wide. Each addressable memory location is a word, and a memory address is a word. Each register is a word.

By itself, a word has no inherent meaning: it is just a sequence of bits. Some instructions operate on a word without regard to what it means: for example, several instructions copy a word from one place to another and it doesn't matter what the word means. Other instructions act on a word assuming that it represents some particular primitive data type. For example, integer arithmetic assumes that the word represents an integer, while address arithmetic assumes that the word represents a natural number.

Sigma16 supports natural numbers, and integers, which are represented as words. It also supports Booleans, which are represented as a bit within a word. Addresses and characters are represented as natural numbers.

4.3.1 Indexing bits in a word

The bits of a word are indexed from right to left, starting with index 0. The least significant (rightmost) bit has index 0, and the most significant bit (leftmost) has index 15.

The notations x_i and `x.i` both mean the bit with index i in the word x, for $0 \leq i \leq 15$. For example, x_0 is the rightmost bit and x_{15} is the leftmost bit. When used in an instruction, a bit index is specified as a 4-bit binary number i such that $0 \leq i \leq 15$.

The following table shows the indices of all the bits in a word. The vertical bars break the word into groups of 4 bits. This grouping corresponds to the representation of the word in hex notation.

$x_{15}\ x_{14}\ x_{13}\ x_{12}$	$x_{11}\ x_{10}\ x_9\ x_8$

$x_7\ x_6\ x_5\ x_4$	$x_3\ x_2\ x_1\ x_0$

4.3.2 Fields

A bit field is a contiguous sequence of bits in a word. It is specified by two numbers: the indices of the rightmost and leftmost bits in the field.

4.3.3 Natural numbers

The natural numbers are $0, 1, 2, \ldots$, so all natural numbers are nonnegative. Sigma16 represents natural numbers as binary, and uses them to represent memory addresses. The binary value of an n bit generic word x is

$$binval(x) = \sum_{0 \leq i < n} x_i * 2^i$$

For a word of 16 bits, natural numbers are restricted to the range from 0 through $2^{16} - 1$; that is, from 0 through 65,535. For a double word (32 bits), natural numbers are restricted to the range from 0 through $2^{32} - 1$; that is, from 0 through 4,294,967,295.

A natural number (and hence a binary number) cannot be negative. If you need numbers that can be negative or positive, you must use an integer. The arithmetic instructions in Sigma16 operate on integers, but address arithmetic is performed in binary.

4.3.4 Integers

Integers are represented using two's complement notation. If the leftmost (most significant) bit of a word is 0, its two's complement value is the same as its binary value. If the leftmost bit is 1, the two's complement value is negative.

Any two's complement number (with one exception) can be negated by inverting all the bits (replace 0 by 1 and vice versa) and then adding 1, discarding any overflow. The sole exception is that if the smallest negative number (-2^{15}) is negated, the result is overflow because the largest positive integer representable in 16 bit two's complement is $2^{15} - 1$.

For example, consider x = 1111 1010. Since the leftmost bit is 1, we know that $x < 0$. We can negate x by inverting the bits, obtaining 0000 0101. Adding 1 gives 0000 0110 which is 6. Since $-x = 6$, we conclude that $x = -6$.

4.3.5 Notations for a word

Assembly language provides several notations for expressing the value of a word.

- A natural number between 0 and 65,535 (2^{16} - 1)
- An integer between -32,768 and 32,767 (-2^{15} and 2^{15} - 1)
- A 4-digit hexadecimal constant, where the digits are 0-9 a-f. The constant is preceded by $, for example $3e8c.

In assembly language programs, hexadecimal constants are written with a preceding $ sign (e.g. $3b2f). This is necessary to avoid ambiguity: 1234 is a decimal number and $1234 is a hexadecimal number, and they denote different words.

Assembly language always requires a $ before a hexadecimal constant. For displays of the machine state, in contexts where there is no ambiguity, the $ may be omitted. For example, the user interface displays register and memory contents as hexadecimal without the leading $. However, the $ is always required for hex constants in assembly language programs.

4.4 Memory

The memory is a hardware array of words that are accessed by address. A memory address is 16 bits wide, and there is one memory location corresponding to each address, so there are 2^{16} = 64k memory locations (1k = 1,024). Each memory location is a 16-bit word.

Instructions specify memory addresses in two parts: the **displacement**, which is a word representing a binary number, and the **index**, which is one of the registers in the register file. For example, a memory address could be specified as $003c[R5]; the displacement is 003c and the index is R5.

When the instruction is executed, the computer calculates the **effective address** by adding the value of the displacement and the value in the index register. If R5 contains 2, then the effective address of $003c[R5] is 003e.

This scheme may seem more complicated than simply specifying the address directly, but it is flexible. If the machine language just gave the address as a single binary number, it would be limited to accessing simple static variables. The effective address mechanism is simple to implement in hardware, as you can see in the digital circuit processor, yet it allows the implementation of static variables, local variables, records, arrays, pointers and linked data structures, jump tables, and more. These techniques are described later.

4.5 Registers

4.5.1 Register file

The **register file** is a set of 16 general registers, each of which holds a 16 bit word. A register is referenced by a 4-bit binary number. In assembly language, we use the notations R0, R1, R2, ..., R9, R10, R11, R12, R13, R14, R15 to refer to the registers. The state of the register file can be written as a table showing the value of each register:

Register	Contents
R0	0000
R1	fffe
R2	13c4
...	...
R14	03c8
R15	0020

Sigma16 is a load/store style architecture. All calculations are carried out in the register file, and explicit load and store instructions must be used to copy data between the memory and the register file.

R0 and R15 are special. The other registers, R1 through R14, may be used for any purpose.

- R0 always contains the constant 0.

It is legal to perform an instruction that attempts to load some other value into R0, but the register will still contain 0 after executing such an instruction. Such an instruction will simply have no lasting effect.

- R15 is the condition code register.

4.5.2 Condition code

Several instructions produce status information: the result of a comparison, whether there was an overflow, etc. This information is automatically loaded into R15, which is also called the *condition code*. The description of each instruction states whether R15 is modified, and what goes into it.

Each bit in the condition code (R15) is a Boolean that specifies whether a specific condition is false (0) or true (1). The bits are indexed from bit 0 (rightmost, or least significant) to bit 15 (the leftmost, or most significant). Condition code bits are sometimes called *flags*. Each flag gives the status of a relation or event. If the flag is True (1) the relation holds or the event has occurred. If the flag is False (0) the relation does not hold, or the event has not occurred.

One way to use flags in the condition code is to control conditional jumps. The jumpc0 and jumpc1 instructions are conditional jumps that use a flag in R15 to decide whether to jump:

- Use jumpc0 to jump if the Boolean is False
- Use jumpc1 to jump if the Boolean is True

Both jumpc0 and jumpc1 take two operands. The first is a natural number from 0 to 15, which gives the index of a flag in the condition code. The second is an address to jump to, specified as displacement[index]. The indices of the flags are given below.

The two conditional jumps (jumpc0 and jumpc1) let you jump depending on either the truth or falsity of a condition. For example, the flag at index 4 denotes integer less than. The fist pair of instructions below jumps if the less than condition holds, and the second pair jumps if the less than condition is false. This is equivalent to jumping if the "greater than or equal" condition holds.

```
cmp     R8,R9
jumpc1  4,xyz[R0] ; if R8 < R9 then goto xyz
\ldots{}
cmp     R8,R9
jumpc0  4,xyz[R0] ; if R8 >= R9 then goto xyz
```

Programs usually use pseudoinstructions rather than the actual machine instructions jumpc0 and jumpc1. For example, the following two instructions are equivalent: the assembler generates exactly the same machine language code for them.

```
jumpc1  4,xyz[R0] ; if R8 < R9 then goto xyz
jumplt  xyz[R0]   ; if R8 < R9 then goto xyz
```

In effect, the pseudoinstruction lets you omit the flag index and instead to indicate the condition you want in the mnemonic. It's easier to remember that less than is indicated by lt than by 4.

Another way to use condition code flags is to save them as Boolean variables and perform logic operations on them. In this case it's best to copy a flag to another register, using copyb. From there you can evaluate logic expressions using andb, orb, xorb, and logicb. It isn't a good idea to keep Boolean variables in R15 for a long time, because many instructions will modify it.

Before doing logic operations on condition code bits, consider whether "short circuit" evaluation, with a separate comparison followed by conditional jump for each relation, is more appropriate.

There is not just one flag for "less than" and another for "greater than". There are separate flags for integers (represented as two's complement) and natural numbers (represented as binary). This is necessary because the relation between two words sometimes depends on the type of the data. For

example, consider the word ffff (all 1 bits). On its own, ffff is just a word of bits and has no inherent meaning.

- If ffff is interpreted as a natural number (i.e. binary), it is positive and has the value 65,535, and ffff $>$ 0000.
- If ffff is interpreted as an integer (i.e. two's complement), then it is negative and has the value -1, and ffff $<$ 0000.

The cmp instruction compares two words, and evaluates five relations between the words: equality, less than for both natural and integer, and greater than for both natural and integer. These evaluations are performed simultaneously, using parallelism in the ALU circuit, and the result of each is recorded in the corresponding flag in the condition code.

```
cmp    R5,R6
jumpc1 3,lessNat[R0]  ; if R5 <Nat R6 then goto lessNat
\ldots{}
cmp    R7,R8
jumpc1 4,LessInt[R0]  ; if R7 <Int R8 then goto lessInt
```

Thus you don't specify the types of the operands in the cmp instruction. Instead, you select the condition code flag that corresponds to the types.

Each flag has a short 1-character name to enable it to be displayed compactly in the user interface. A naming convention is that flags for integers (two's complement) have lower case letters, while flags for natural numbers (binary) have upper case letters. For example:

- l means $<$ for integers
- L means $<$ for natural numbers
- g means $>$ for integers
- G means $>$ for natural numbers

Equality is the same regardless of type. If two words consist of exactly the same bits, then they have the same value as integers, natural numbers, characters, addresses, and for any other possible type as well. Therefore there is only one flag for equality, and its symbol is $=$.

Register R15 is also called the *condition code*. Some instructions modify a bit in the condition code to indicate the results of comparisons, overflow, carry, other special data, and various errors. Instructions that operate on registers can access R15 as usual. There are also some instructions that implicitly access R15 without specifying it (for example, jumpc0 and jumpc1). Since many instructions modify the condition code as a side effect, programs should not use R15 to hold variables.

The following table lists all the condition code flags.

- Index is the bit position of the flag in the condition code. Bits are numbered from right to left, starting with 0.
- Symbol: the graphical user interface shows the symbol for each bit that is 1. For example, if G is displayed after a comparison of two words, this means that $>$ holds if the words are interpreted as natural numbers but not if they are interpreted as integers. However, if G$>$ is displayed, the comparison result is greater than for both integer and two's complement representation.
- Meaning: Description using English or mathematical notation
- Hex: The value of the condition code word assuming the given flag is true and all others are false. If several flags are true, the condition code register will be the logic or of the hex values of the true flags. For example, suppose the words 00a3 is compared with 00c5. The comparison result is L (less than for naturals) and also l (less than for integers). So the condition code will contain 0008 + 0010 = 0018.

Table: Condition code flags

Index	Symbol	Meaning	Hex
0	> g	> Int	0001
1	G	> Nat	0002
2	=	=	0004
3	L	< Nat	0008
4	< l	< Int	0010
5	v	Int overflow	0020
6	V	Nat overflow	0040
7	C	Carry	0080
8	S	Stack overflow	0100
9	s	Stack underflow	0200
10		reserved	0400
11		reserved	0800
12		reserved	1000
13		reserved	2000
14		reserved	4000
15		reserved	8000

There is an exception for division by zero, but no corresponding flag in the condition code. The reason is that the div instruction places the remainder in R15, so the condition code isn't available to represent division by 0. You can use an interrupt to detect division by 0, and you can test explicity for division by 0 by using jumpz specifying the register containing the divisor before executing the div instruction.

1. The ordinary general registers: R1 through R14

   Registers R1 through R14 have no special properties defined by the machine. A program can use them for any purpose.

   However, there are some programming conventions that use certain registers for special purposes. The hardware does not enforce, or even know about, these conventions, and you do not have to follow the conventions in programming. However, it is necessary to obey the conventions in order to use the standard software libraries in your program. See the section on Programming for a discussion of these standard usage conventions.

4.5.3 Instruction control registers

The instruction control registers enable the processor to keep track of the state of the running program. These registers are rarely used directly by the machine language program, but they are essential for keeping track of the execution of the program, and some instructions use them directly.

1. pc – program counter

   The PC (program counter) register contains the address of the next instruction to be executed (*not* the address of the instruction currently being executed). The name is illogical, but "program counter" is the traditional name and Sigma16 sticks with standard terminology.

2. ir – instruction register

   The IR contains the instruction that is currently executing. If the instruction is RRR format, the entire instruction is in the IR. The other instruction formats require two words; the first word is in the IR and the second word is in the ADR register.

3. adr – address register

   If the instruction currently executing is RRR format, the ADR register is not used, and its content is just whatever was left over from previous instructions.

   If the instruction currently executing is RX or EXP format, the ADR register contains the second word of the instruction. Furthermore, for RX instructions the effective address is calculated and replaces the second word of the instruction.

4. status – system status register

   The status register contains flags that specify the current execution mode of the processor.

Index	Symbol	Meaning	Hex
0	U/S	0: user state. 1: system state	0001
1	E	1: interrupts enabled	0002

If the processor is in system state, any instruction can be executed. If it is in user state, executing a privileged instruction causes an interrupt.

When the system is started by booting a program, the status is set to system state. If the program is just a standalone program (such as the simple examples), you can ignore the system state. If an operating system has been booted, the OS should put the processor into user state when it gives a time slice to a user process.

If the status register indicates that interrupts are enabled, and an interrupt request is set in the request register, than an interrupt occurs when the current instruction is completed. If interrupts are not enabled, then interrupt requests are ignored.

4.6 Interrupts

An interrupt is a jump performed automatically by the machine in response to an event, not by executing a jump instruction. Interrupts are used for error detection and recovery, input/output, concurrency, memory protection, and virtual memory. Most of these features are implemented by an operating system kernel.

There are two kinds of event that can trigger an interrupt:

- An error that occurs while executing an instruction. For example, an arithmetic instruction could produce a result that is too large to represent in a word (this is called overflow). Sometimes this is called an *exception*, and sometimes it is called an interrupt.

- An external event that has nothing to do with the running program, but which needs to be handled. For example, an I/O device may request an interrupt when it has completed an I/O operation. This is always called an interrupt.

There are several types of interrupt. Each type has a unique index. The interrupt indices serve two purposes: they determine the interrupt priority, and they give the bit index for the interrupt type in the mask and request registers. The following table shows each type of interrupt and the bit index of its flag.

Table: Interrupt Indices

Index	Symbol	Meaning	Hex
0	time	timer interrupt	0001
1	sf	segmentation fault	0002
2	sto	stack overflow	0004
3	stu	stack underflow	0008
4	trap	user trap	0010
5	tco	integer overflow	0020
6	bo	natural overflow	0040
7	zdiv	divide by 0	0080

Each type of interrupt has a priority which is determined by its index: a lower index indicates higher priority. Thus the timer interrupt has the highest priority, and a stack overflow has higher priority than integer overflow.

Interrupts take place only between execution of instructions. When one instruction has finished, the processor checks for interrupts before going on to the next instruction. This consists of the following steps:

1. If interrupts are not enabled, the check terminates and the next instruction is executed normally.

2. If interrupts are enabled, and there is an interrupt request, and the corresponding mask bit is enabled, an interrupt will take place. If several types of interrupt meet these conditions, the one with highest priority (lowest index) is selected. Let i be the index of the type of interrupt that was selected.

3. The processor clears the request bit for the interrupt in the request register, it copies the status register into rstat, and it copies the pc register into the rpc register. Also, it loads mem[vect+2*i] into the pc register, and it and sets flags in the status register to indicate system state and interrupts disabled.

4. If the timer is running, it is stopped.

5. Now the interrupt has completed, and the next instruction is executed normally. Since the pc has been modified, the machine has jumped to a different location.

This algorithm is performed entirely by the processor; none of it involves software. Although it is described as a sequence of steps, many of these steps can be performed in parallel, because digital circuits are inherently parallel. If an interrupt does not take place, the time required for the check is minimal (less than one clock cycle), so checking for interrupts does not signicantly slow down the processor speed.

4.6.1 mask - interrupt mask register

The mask register specifies which types of interrupt are allowed to occur. It contains a flag for each type of interrupt; if the flag is 1 then the corresponding type of interrupt can take place. If the mask flag is 0, then that type of interrupt will not occur even if it is requested.

4.6.2 request - interrupt request register

When an interrupt is requested, the corresponding bit is set in the request register. When the next instruction is executed, the interrupt occurs if interrupts are enabled and the corresponding mask bit is 1.

4.6.3 rstat register

When an interrupt occurs, the value of the status register is copied into rstat.

4.6.4 rpc register

When an interrupt occurs, the value of pc is copied into rpc. This is necessary to enable the operating system to resume the interrupted program.

4.6.5 vect register

The interrupt vector register contains the address of a jump table. This is an array of jumps to interrupt handlers.

4.7 Memory management registers

(Will be implemented in future version)

5 Instruction set

5.1 Instruction representation

Instructions are represented in the memory of the computer using words, just like all other kinds of data. From the programmer's perspective, an instruction is like a simple statement in a programming language. From the circuit designer's perspective, instructions must be executed using logic gates, and the specific way it is represented as a word of bits is important.

An instruction specifies several pieces of information. For example, the instruction `add R1,R2,R3` says four things: it's an addition, the operands come from R2 and R3, and the result goes into R1. Therefore to represent an instruction we need to organize a word as a collection of several **fields**. A field is a portion of a word that contains a small number (for example, a register number). Each field gives one specific piece of information about the instruction.

The particular scheme for describing an instruction as a collection of fields is called an **instruction format**. Like most computers, Sigma16 has a small number of instruction formats and a larger number of instructions. Instruction representation is key to both system software (compilers generate instructions) and computer hardware (a processor is a digital circuit that decodes and executes instructions).

The core architecture (the simplest part of the system) uses two instruction formats: the **RRR format** for instructions that perform calculations in the registers, and the **RX format** for instructions that refer to a memory location.

The advanced parts of the architecture provide additional instructions, most of which are represented with the **EXP format**. The name EXP stands simultaneously for **expansion** (because it provides for many additional instructions) and **experimental** (because it allows for experimentation with the design and implementation of new instructions).

Each instruction format has a fixed size, which is the number of words used to represent any instruction of that format:

- RRR instructions are one word
- RX instructions are two words
- EXP instructions are two words

The first word of every instruction (for all instruction formats) contains the following fields:

- op (bits 15-12) 4-bit opcode, determines instruction format
- d (bits 11-8) 4-bit destination
- a (bits 7-4) 4-bit operand a
- b (bits 3-0) 4-bit operand b, or expanded opcode for RX

op	d	a	b
15-12	11-8	7-4	3-0

The format of every instruction is determined by the op field. Since this field contains 4 bits, it can specify 16 values:

- $RRR.$ If the op field is between 0 and 12 (hex 0 to hex c), the instruction is RRR format, and there are 13 RRR instructions.
- **Reserved.** An op field value of 13 is reserved for future expansion of the architecture.
- **EXP.** If the op field is 14 (hex e) the instruction is EXP format and has a secondary opcode in the a and b fields.
- **RX.** If the op field contains 15 (hex f) the instruction is RX format with a secondary opcode in the b field.

Since an RRR instruction is one word, it contains just the op, d, a, b fields. If an instruction is RX or EXP format, it has a second word whose format is described below in the sections on RX and EXP.

5.1.1 RRR format

The RRR format is used for instructions that perform calculations in the registers, without using memory.

An instruction in RRR format is one word containing four 4-bit fields called op, d, a, b. Each field contains 4 bits representing a binary number between 0 and 15.

op	d	a	b

- op (bits with index 15 to 12) is the operation code, usually called *opcode*. This determines the operation to be performed. If the opcode is between 0 and 12 it specifies an RRR instruction. An opcode greater than 12 indicates an *expanding opcode*: the instruction is not RRR but one of the other formats, and it has a secondary opcode that specifies precisely which instruction it is. This is explained in the sections on RX and EXP formats.
- d (bits 11 to 8) is the *destination register*; the register where (in most cases) the result will be loaded.
- a (bits 7 to 4) is the register containing the first operand.
- b (bits 3 to 0) is the register containing the second operand.

In most cases, an RRR instruction takes two operands in registers specified by the a and b fields and produces a result which is loaded into the register specified by the d field.

A typical example of an RRR instruction is add R4,R13,R2, which adds the contents of registers R13 and R2, and loads the result into R4. It's equivalent to R4 := R13 + R2. The opcode for add is 0, so the machine language code for this instruction is 04d2.

5.1.2 RX format

The RX format is used for instructions that use a memory location. There are two operands: one is a register and the other is the address of a word in memory. Typical RX instructions are loads, stores, and jumps. For example, a load has an address operand and a destination register operand; it loads the word in memory at the specified address into the destination register.

A memory address is just a natural number. However, the second operand of an RX instruction, a memory address, is specified by two pieces: an index register and a natural number called the displacement.

The name of the format describes briefly these two pieces: a register (R) and an indexed memory address (X).

The instruction consists of two consecutive words. The first has the same format as an RRR instruction, with four fields: op, d, sa, sb. The second word is a single 16-bit binary number, and is called the displacement.

An RX instruction is represented by two words. The first word has the same fields as an RRR instruction:

- The op field (bits 15-12) is 15 (hex f). This indicates that the instruction has format
- The d field (bits 11-8) is the first operand, the destination register.
- The a field (bits 7-4) is called the index register, and is part of the specification of the effective address.
- The b field (bits 3-0) is the secondary opcode.

The second word contains one field, the 16 bit displacement.

- disp (displacement) is the second word of the instruction

disp
15-0

ea (effective address) = displacement + r[a]

The memory address is specified in two parts: an index register and the displacement. The index register is specified in the sa field. In assembly language, the notation used is number[reg], where the number is the value of the displacement, and the reg is the index register. Thus $20b3[R2] means the address has displacement $20b3 and the index register is R2.

When the machine executes an RX instruction, it begins by calculating the effective address. This is abbreviated "ea", and its value is the sum of the displacement and the contents of the index register.

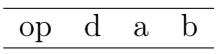
RX instructions are represented in two words, and they use an "expanding opcode". That is, the op field of the first word of the instruction contains the constant f (the bits 1111) for every RX instruction, and the sb field is used to hold a secondary opcode indicating which RX instruction it is.

The register operand is specified in the d field. For several RX instructions, this is indeed the destination of the instruction: for example, load places data into Rd. However, a few RX instructions use the d field differently (see, for example, the conditional jump instructions).

The memory address is specified using the sa field and the displacement, which is the entire second word of the instruction.

The RX format is used for instructions that access the memory. There are two operands: a memory address and a register. The memory address is specified using two fields: a constant (called the displacement) and a register (called the index register).

An RX instruction consists of two words. The first has the same format as RRR. The op field is 15, which means "this instruction has RX format". A secondary operation code is needed to specify which RX instruction this is, this is given in the b field. The d field is the destination register, and the a field is the index register. There is only one operand register for RX instructions, since the b field is needed for the secondary operation code.



The second word consists of one 16-bit field called the displacement (abbreviated as disp).

The index register Rb and the displacement together specify the *effective address*. The assembly language syntax for the effective address is `disp[Rb]`, and its value is disp + Rb. For example, suppose R4 contains 7 when the following instruction is executed:

```
load  R2,5[R4]
```

The address operand is `5[R4]`, where the displacement is 5 and the index is R4. When the instruction executes, the effective address is 5+R4 = 5+7 = 12, or `$000c`.

The displacement is a constant given in the instruction, but the index register is variable. Since R0 always contains 0, the effective address for `disp[R0]` is the value of `disp`.

The displacement is represented in binary, and the effective address is calculated in binary, not in two's complement. Thus the effective address of `ffff[R0]` is the (positive) address of the last word in memory – it isn't a negative number.

5.1.3 EXP format

The EXP instructions provide more complex operations, and they belong to the Standard architecture. (The Core architecture uses only RRR and RX). An EXP instruction consists of two words.

The first word has the same op and d fields as RRR and RX. The op field contains 14 (hex e), which indicates that the instruction is EXP format. The a and b fields are treated as one 8-bit natural number, which is the secondary operation code. This provides for the possibility of up to 256 EXP format instructions, which enables new experimental instructions to be defined.

op	d	ab

The second word contains for 4-bit fields. Each of these may contain either a register number or a short 4-bit number, depending on the instruction.

e	f	g	h

- e (bits 15-12) 4-bit operand
- f (bits 11-8) 4-bit operand
- g (bits 7-4) 4-bit operand
- h (bits 3-0) 4-bit operand

e	f	g	h
15-12	11-8	7-4	3-0

5.1.4 Summary of instruction formats

Format	Size	Opcode	Operands	Example
RRR	1	op	d,a,b	add Rd,Ra,Rb
RX	2	op,b	d,a,disp	load Rd,disp[Ra]
EXP	2	op,ab	d,e,f,g,h	save Rd,Re,gh[Rf]

Although there are three machine language instruction formats (RRR, RX, EXP), there is a larger set of assembly language statement formats, because there are special syntaxes for some instructions, and there are assembler directives that aren't instructions at all. Thus there are two kinds of format: the machine instruction formats, and the assembly language instruction statement formats. The assembly language formats are described later.

5.1.5 Notation for machine language

We usually write instructions in assembly language (sub R3,R12,R9) but the computer executes machine language (13c9). There is a precise relationship between these two notations for an instruction. By writing assembly language, you determine the exact value of every bit in every word of a machine language program. This section explains how these two notations for an instruction are related.

When each instruction is described below, an example is given showing a typical use of the instruction, along with the general form of the assembly language instruction.

The general form uses the machine language field names to show where each piece of the instruction goes in the machine code. The instruction format is also given, along with any constant fields.

As an example, the logicu instruction is EXP format, so its machine language representation consists of two words with the following fields:

op	d	a	b
e	f	g	h

The primary opcode is e, which goes in the op field. The secondary opcode is 14, which goes in the ab field. Thus op=e, a=1, and b=4 (here, a dollar sign is used before the hex constants to distinguish them from the field names).

$e	d	$1	$4
e	f	g	h

and the For example, here is the general form of the logicu instruction:

```
logicu  Rd,e,Rf,g,h     ; Rd.e := h (Rd.e, Rf.g)
EXP op=\#e ab=\$14
```


```
logicu  Rd,e,Rf,g,h     ; Rd.e := h (Rd.e, Rf.g)
EXP op=\#e ab=\$14
```


The format is EXP, so the instruction has all the fields of the EXP format: op, d, ab, e, f, g, h. There are two constant fields: the primary opcode (op) is

`$e`, and the secondary opcode (ab) is `$14`. The values of the other fields are given in the general form of the assembly language. The operands Rd and Rf refer to registers, and those register numbers go into the d and f fields of the instruction. The e, g, and h fields are given as numbers in the assembly language. (The assembly language convention is to give constants in decimal notation, not hexadecimal.)

Example:

```
logicu  R7,5,R2,13,xor ; R7.5 := R7.5 xor R2.13
e714 52d6
```

5.2 Arithmetic

5.2.1 add

The instruction `add Rd,Ra,Rb` has operands `Ra` and `Rb` and destination `Rd`. It fetches the operands `Ra` and `Rb`, calculates the sum `Ra` + `Rb`, and loads the result into the destination `Rd`. The effect is `Rd := Ra + Rb`. For example, `add R5,R12,R2` performs `R5 := R12 + R3`.

As well as setting the destination register, the add instruction sets or clears the overflow and carry bits in the condition code (which is R15). The other bits in R15 are not changed.

If the destination register is R15, the sum is placed in R15, and the overflow and carry flags are discarded.

The add instruction is RRR format with opcode=0. Given destination Rd and operands Ra and Rb (where d, a, b are hex digits), add Rd,Ra,Rb is reprseented by 0dab.

assembly language	add Rd,Ra,Rb
instruction format	RRR
opcode	0
machine language	$0 d a b
effect	R[d] := R[a] + R[b]
condition code	vVC

Examples:

Code	Assembly	Effect
062c	add R6,R2,R12	; R6 := R2 + R12
0d13	add R13,R1,R3	; R13 := R1 + R3

A field is a consecutiave sequence of bits within a word. A field is specified with the index of the leftmost bit in the field, along with the size of the field. For example, the field in x with index 9 and size 3 consists of the bits x.9 x.8 x.7.

```
add R1,R2,R3    ; R1 := R2 + R3
```

The instruction add Rd,Ra,Rb has operands Ra and Rb and destination Rd. It fetches the operands Ra and Rb, calculates the sum Ra + Rb, and loads the result into the destination Rd. The effect is Rd := Ra + Rb. For example, add R5,R12,R2 performs R5 := R12 + R3.

The add instruction is RRR format with opcode=0. Given destination Rd and operands Ra and Rb (where d, a, b are hex digits), add Rd,Ra,Rb is reprseented by 0dab.

The add instruction can be used for both binary addition (on natural numbers) and for two's complement addition (on signed integers).

- 16-bit natural numbers are unsigned integers 0, 1, 2, ..., 65535. If two natural numbers are added, the result is a natural number (the result cannot be negative). If the result is 65536 or larger, it cannot be represented as a 16 bit binary number. If this happens, the destination register is set to the lower 16 bits of the true result, and the binary overflow flag is set in the Condition Code.

- 16-bit two's complement numbers are signed integers -32999?, ..., -1, 0, 1, ..., 32???. If two signed integers are added, the result is a signed integer. If the result is less than -32000 or greater than 32000, then the result cannot be represented as a 16 bit two's complement number. If this happens, the destination register is set to the lower 16 bits of the true result, and the two's complement overflow flag is set in the Condition Code. Furthermore, the overflow flag is set in the req register. If interrupts are enabled and the overflow flag is 1 in the mask register, then an interrupt will occur immediatelhy after the add instruction executes.

The add instruction can be used for both binary addition (on natural numbers) and for two's complement addition (on signed integers).

- 16-bit natural numbers are unsigned integers 0, 1, 2, ..., 65535. If two natural numbers are added, the result is a natural number (the result cannot be negative). If the result is 65536 or larger, it cannot be represented as a 16 bit binary number. If this happens, the destination register is set to the lower 16 bits of the true result, and the binary overflow flag is set in the Condition Code.

- 16-bit two's complement numbers are signed integers -32999?, ..., -1, 0, 1, ..., 32???. If two signed integers are added, the result is a signed integer. If the result is less than -32000 or greater than 32000, then the result cannot be represented as a 16 bit two's complement number. If this happens, the destination register is set to the lower 16 bits of the true result, and the two's complement overflow flag is set in the Condition Code. Furthermore, the overflow flag is set in the req register. If interrupts are enabled and the overflow flag is 1 in the mask register, then an interrupt will occur immediatelhy after the add instruction executes.

5.2.2 sub

Example: sub R1,R2,R3 ; R1 := R2 - R3

This instruction is similar to add; the only difference is that it calculates R2-R3 and places the result in R1. The effect on the condition code is the same as for add.

assembly language	sub Rd,Ra,Rb
instruction format	RRR
opcode	1
machine language	$1 d a b
effect	R[d] := R[a] - R[b]

The instruction sub Rd,Ra,Rb has operands Ra and Rb and destination Rd. It fetches the operands Ra and Rb, calculates the difference Ra - Rb, and loads the result into the destination Rd. The effect is Rd := Ra - Rb. For example, sub R5,R12,R2 performs R5 := R12 - R3.

The sub instruction is RRR format with opcode=1.

Code Assembly Effect —— ————- ———— 162c sub R6,R2,R12 ; R6 := R2 - R12 1d13 sub R13,R1,R3 ; R13 := R1 - R3 In addition to setting the destination register, the sub instruction sets several bits in the condition code R15 and may set a bit in the req register.

——— ———— R15.ccG result $>$ 0 (binary) R15.ccg result $>$ 0 (two's complement) R15.ccE result = 0 R15.ccl result $<$ 0 (two's complement) R15.ccV overflow (binary) R15.CCv overflow (two's complement) R15.CCc carry output R15.ccf logicc instruction function result ——— ———— —

5.2.3 mul

Example: mul R1,R2,R3 ; R1 := R2 * R3

The multiply instruction mul Rd,Ra,Rb calculates the integer (two's complement) product of the operands Ra and Rb, and places the result in the destination register Rd. The mul instruction does not produce the natural (binary) product.

assembly language	mul Rd,Ra,Rb
instruction format	RRR
machine language	[$2 d a b]
effect	reg[d] := reg[a] * reg[b]

If the magnitude of the product is too large to be representable as a 16 bit two's complement integer, this is an overflow. If overflow occurs, the integer overflow bit is set in the condition code (F15) and the integer overflow bit is also set in the interrupt request register (req), and the lower order 16 bits of the product are loaded into Rd.

——— ———— R15.ccg result $>$ 0 (two's complement) R15.ccE result = 0 R15.ccl result $<$ 0 (two's complement) R15.CCv overflow (two's complement) R15.CCc carry output ——— ————

5.2.4 div

Example: div R1,R2,R3 ; R1 := R2 / R3, R15 := R2 rem R3

Unlike the other arithmetic operations, the divide instruction div Rd,Ra,Rb produces two results: the quotient Ra / Rb and the remainder Ra rem Rb. It loads the quotient into the destination register Rd, and the remainder is loaded into R15.

assembly language	div Rd,Ra,Rb
instruction format	RRR
opcode	3
machine language	$3 d a b
effect	reg[d] := reg[a] / reg[b], reg[15] := reg[a] rem reg[b]

If the destination register Rd is R15, then the quotient is placed in R15, and the remainder is discarded.

The divide instruction doesn't set the condition code, since R15 is used for the remainder. Therefore there is no condition code bit to indicate division by 0. However, it is easy for a program to detect a division by 0.

- (Explicit test for error) The program can compare the divisor with 0 before or after executing the divide instruction, and jump to an error handler if the divisor is 0. This is similar to testing the condition code after an add, sub, or mul instruction, but it does require two instructions: a compare followed by a conditional jump. For example:

```
div    R1,R2,R3       ; R1 := R2/R3, R15 := R2 rem R3
cmp    R3,R0       ; Did we divide by 0?
jumpeq zeroDivide[R0] ; If yes, handle error
```

- (Exception) The program can detect division by 0 using an interrupt. To do this, enable interrupts and enable the interrupt mask for division by 0. See the section on Interrupts. This approach does not require a compare or jump instruction for each division.

5.2.5 cmp

The compare instruction `cmp Ra,Rb` compares the values in the operand registers `Ra` and `Rb`, and then sets flags in the condition code (R15) to indicate the result. The notation R15.i means bit i in R15; thus R15.0 is the leftmost bit of R15. The instruction performs both natural number comparison (binary) and integer comparison (two's complement). The resulting flags are

- binary less than (L) in R15.0
- two's complement less than ($<$) in R15.1

- equal in R15.2
- binary greater than (G) in R15.3
- two's complement greater than (>) in R15.4

assembly language	cmp Rd,Ra,Rb
instruction format	RRR
opcode	4
machine language	$4 d a b
effect	reg[d] := reg[a] / reg[b]

The result of a cmp instruction can be used to control a conditional jump. The jumpc0 instruction jumps is a specified bit of R15 is 0, and the jumpc1 instruction jumps if a specified bit is 1.

Pseudoinstructions provide the most common cases; for example jumple jumps if the condition code indicates that a comparison produced either integer less-than or equal. A common pattern is a cmp followed by a jump pseudoinstruction, for example:

```
cmp     R4,R9       ; compare R4 with R9
jumpgt  abc]R0]     ; if R4 > R9 then goto abc
```

5.2.6 addc

The addc instruction performs a binary addition with carry propagation. It adds the two operand registers and the carry bit in the condition code register, R15. The sum is loaded into the destination register Rd and the carry output is written back into the carry bit, overwriting its previous value. Overflow is not possible with this instruction.

5.2.7 muln

```
muln   Rd,Ra,Rb
```

The muln instruction calculates the product of two natural numbers in Ra and Rb. The result is 32 bits; the leftmost 16 bits (the most significant part) is loaded into R15, and the rightmost 16 bits (the least significant part) is loaded into Rd. If Rd is R15, the most significant part is discarded.

instruction format	RRR
machine language	[$4 d a b]
effect	reg[d] := reg[a] / reg[b]
assembly language	cmp Rd,Ra,Rb

5.2.8 divn

```
divn   Rd,Ra,Rb
```

The divn instruction divides two natural numbers: dividend / divisor. All the numbers – numerator, denominator, quotient, remainder – are natural numbers represented in binary.

instruction format	RRR
machine language	[$4 d a b]
effect	reg[d] := reg[a] / reg[b]
assembly language	divn Rd,Ra,Rb

- The dividend is a 32 bit natural number; its leftmost 16 bits are in R15 and the rightmost 16 bits are in Ra. Thedenominator is in Rb.
- Two results are produced: a 32-bit quotient anda 16-bit remainder.
- The leftmost 16 bits of the quotient are placedin R15 (replacing the leftmost part of the dividend). The rightmost16 bits of the quotient are placed in Rd.
- The remainder is placed in Ra, overwriting the least significant half of the dividend operand

5.3 Accessing memory

A memory address is a 16-bit binary number. Instructions don't specify addresses directly; they specify an address with two components: a **displacement** and an **index**, written as "displacement[index]". The displacement is a 16 bit constant, and in assembly language it may be given as a decimal integer, a hexadecimal word, or a label. The index is a register. For example,$0c45[R5] has a displacement of 0c45 and an index of R5.

When an instruction executes, the machine takes the displacement and index and calculates the **effective address**. This is defined to be the binary sum of the displacement and the curent value in the index register. In the example above, if R5 contains 3, then the effective address of $0c45[R5] is $0c48.

If you just want to specify an address **a** in an instruction, this can be written as "a[R0]". Since R0 contains the constant 0, the effective address is just **a**.

5.3.1 lea

The **load effective address** instruction **lea Rd,disp[Ra]** calculates the effective address of the operand disp[Ra] and places the result in the destination register Rd. The effective address is the binary sum disp+Ra.

assembly language	lea Rd,disp[Ra]
opcode	f, 0
instruction format	RX
machine language	$f d a $0, disp
effect	reg[d] := disp+reg[a]

5.3.2 load

The **load** instruction **load Rd,disp[Rx]** calculates the effective address of the operand disp[Rx] and copies the word in memory at the effective address into the destination register Rd. The effective address is the binary sum disp+Rx.

assembly language	load Rd,disp[Ra]
instruction format	RX
opcode	f, 1
machine language	$f d a $1, disp
effect	reg[d] := mem[disp+reg[a]]

Examples

```
load  R12,count[R0]   ; R12 := count
load  R6,arrayX[R2]   ; R6 := arrayX[R2]
load  R3,\$2b8e[R5]    ; R3 := mem[2b8e+R5]
```

5.3.3 store

The **store** instruction **store Rd,disp[Rx]** calculates the effective address of the operand disp[Rx]. The value of the destination register Rd is stored into memory at the effective address.

assembly language	store Rd,disp[Ra]
instruction format	RX
opcode	f, 2
machine language	$f d a $2, disp
effect	mem[disp+reg[a]] := reg[d]

Store copies the word in the destination register into memory at the effective address. This instruction is unusual in that it treats the "destination register" as the source of data, and the actual destination which is modified is the memory location.

Most instructions take data from the rightmost operands and modify the leftmost destination, just like an assignment statement (x := y+z). However, the store instruction operates in the opposite direction. The reason for this has to do with the circuit design of the processor. Although the "left to right" nature of the store instruction may look inconsistent from the programmer's point of view, it actually is more consistent from the deeper perspective of circuit design.

Examples

```
store  R3,\$2b8e[R5]
store  R12,count[R0]
store  R6,arrayX[R2]
```

5.3.4 Stacks

Three instructions (push, pop, top) support operations on a stack represented as an array of contiguous elements, where the stack grows from lower to higher addresses. These instructions provide safe operations: they never overwrite memory outside the stack, and they indicate stack underflow and overflow by setting the condition code and optionaly performing an exception.

A stack is represented by three addresses, which are provided to the push, pop, and top instructions in registers:

- The *stack base* is the address of the first word allocated for the stack.
- The *stack limit* is the address of the last word allocated for the stack.
- The *stack top* is the address of the stack element that was pushed most recently.

Although three addresses are required to characterise the state of a stack, each individual stack instruction (push, pop, top) requires only two of those addresses. These are supplied as the Ra and Rb operands, while Rd is used to supply or receive the data value.

The maximum number of elements the stack may contain is *stack limit* - *stack base* + 1. Normally, *stack limit* is greater than *stack base*. If they are equal, there is only one word allocated for the stack (which is generally not useful), and if *stack base* > *stack limit* then no memory at all is allocated and every stack operation will signal an underflow or overflow error.

If the stack is not empty, then *stack top* is the address of the top element in the stack. If the stack is empty, then *stack top* must be *stack base* - 1.

A stack can be created and initialized by allocating a region of memory, setting *stack base* to the first word and *stack limit* to the last word, and setting *stack top* to *stack base* - 1.

5.3.5 push

The push instruction pushes an element onto a stack. It is RRR format, and its general form is:

```
push   Rd,Ra,Rb
```

- Rd = *stack data*: value to be pushed, unchanged
- Ra = *stack top*: incremented unless stack was full
- Rb = *stack limit*: unchanged
- R15 condition code indicates stack overflow
- System interrupt request register indicates stack overflow

This instruction pushes the word in Rd onto a stack with *stack top* in Rd and *stack limit* in Rb, provided that the stack is not full. The push stores the data word in Ra into memory and increments *stack top* Rd. If the stack is full, nothing is stored into memory and a stack overflow error is indicated in the condition code and interrupt request registers; an interrupt will occur if interrupts are enabled and the stack mask bit is set. The operational semantics is:

```
if Ra < Rb
  then Ra := Ra + 1; mem[Ra] := Rd
  else R15.sovfl := 1, req.sovfl := 1
```

If Rd = Rb this means the stack completely fills the region of memory allocated for the stack, and there is no space to store a new element. In this case, the push instruction does not store Ra, it doesn't modify Rd, it doesn't modify memory outside the block, and it doesn't overwrite data in the stack. Instead, the instruction indicates a stack overflow by setting the sovfl (stack overflow) bit in the condition code (R15), and it also sets the stack fault bit in the interrupt request register. If interrupts are enabled and the stack fault bit is set in the interrupt mask register, then an interrupt will occur after the push instruction completes. There will be no interrupt if interrupts are disabled, or the stack fault bit is not set in the mask register.

5.3.6 pop

The push instruction removes an element onto a stack and returns it. The instruction is RRR format, and its general form is:

```
pop    Rd,Ra,Rb
```

- Rd = *stack data*: destination for the popped stack element
- Ra = *stack top*: decremented unless stack was empty
- Rb = *stack base*: unchanged
- R15 condition code indicates stack underflow
- System interrupt request register indicates stack underflow

This instruction pops the word from a stack with *stack top* in Ra and *stack base* in Rb, provided that the stack is not empty. The pop loads the top element of the stack into Rd and decrements *stack top*. If the stack is empty, *stack top* is not decremented, Rd is not modified, and a stack underflow error is indicated in the condition code and interrupt request registers; an interrupt will occur if interrupts are enabled and the stack mask bit is set. The operational semantics is:

```
if Ra >= Rb
  then Rd := mem[Ra]; Ra := Ra - 1
  else R15.suvfl := 1, req.suvfl := 1
```

5.3.7 top

The top instruction returns the top element on a stack but does not remove it. The instruction is RRR format, and its general form is:

```
top    Rd,Ra,Rb
```

- Rd = *stack data*: destination for the top element of the stack; unchanged if stack is empty
- Ra = *stack top*: unchanged
- Rb = *stack base*: unchanged
- R15 condition code indicates stack underflow
- System interrupt request register indicates stack underflow

This instruction loads the element at *stack top* into Rd, provided that the stack is not empty. If the stack is empty, Rd is not modified and a stack underflow error is indicated in the condition code and interrupt request registers; an interrupt will occur if interrupts are enabled and the stack mask bit is set. The operational semantics is:

```
if Ra >= Rb
  then Rd := mem[Ra]
  else R15.suvfl := 1, req.suvfl := 1
```

5.3.8 Stack frames

When a program calls a procedure it is usually necessary to save the state of the caller in a data structure called a stack frame. (This is not necessary during a "tail call".) The stack frame is pushed onto the execution stack. When the procedure returns, the contents of the stack frame need to be loaded back into the registers. These operations can be performed by ordinary store instructions (for procedure call) and load instructions (for procedure return). However, this often requires a significant number of instructions.

The `save` and `restore` instructions transfer a block of data between registers and memory, making it easier to use stack frames. These instructions are analogous to `store` and `load`, but they store or load multiple words, not just an individual word.

- **save** stores a sequence of adjacent registers into a block of contiguous memory locations.
- **restore** is the opposite: it loads the block of memory into the registers.

For both instructions, the sequence of registers is specified by giving the first and last register. The starting address of the memory block is specified by an effective address of the form `offset[Reg]`, where Reg is any register (e.g. R4, R13, etc) and `offset` is a number between 0 and 255.

Normally $d \leq e$, in which case the number of registers to be saved is $e - d + 1$. If $d > e$, the instruction will do nothing. The base register R_f should not lie within the group of registers to be saved or restored.

The first register to be saved is R_d, and the last register to be saved is R_e. The instruction always stores at least one register. If $d = e$, for example `save R5,R5,0[R14]` then only R5 is stored. If $d < e$ then the register numbers wrap around For example,

There is an important restriction with `save` and `restore`: the displacement is limited to a small natural number between 0 and 255. (Recall that for load and store, the displacement can be as large as 65,535.) The reason for this is that the save and restore instructions are EXP format and the

offset is represented by an 8-bit field (whereas load and store are RX format and the displacement is a 16-bit word).

The instruction is EXP format, and the offset is limited to 8 bits, because it is specified in the **gh** field, which is the rightmost 8 bits of the second word of the instruction. The secondary opcode is 9, which is in the ab field of the first word of the instruction.

5.3.9 save

The save instruction performs a sequence of stores, which copy the contents of a sequence of registers into consecutive memory locations.

- $\mathsf{mem}[R_f + gh] := R_d$
- $\mathsf{mem}[R_f + gh + 1] := R_{d+1}$
- $\mathsf{mem}[R_f + gh + 2] := R_{d+2}$
- ...
- $\mathsf{mem}[R_f + gh + e - d] := R_e$

assembly language	save Rd,Re,gh[Rf]
instruction format	EXP
opcode	E, 0A
machine language	E d 0A, e f gh
effect	$M[R_f + gh] := R_d$ $M[R_f + gh + 1] := R_{d+1}$ $M[R_f + gh + 2] := R_{d+2}$ ... $M[R_f + gh + e - d] := R_e$

For example, the following instruction will save registers R3 through R9 into memory starting at address 6 + R13. Its machine language code is e30b 9d06.

```
save   R3,R9,6[R13]
```

This instruction is equivalent to a sequence of store instructions:

```
store   R3,6[R13]
store   R4,7[R13]
store   R5,8[R13]
store   R6,9[R13]
store   R7,10[R13]
store   R8,11[R13]
store   R9,12[R13]
```

As this example shows, using `save` and `restore` can make procedure calls more concise and readable as well as less error-prone.

5.3.10 restore

This `restore` instruction has the same operands as `save`, and it performs a sequence of loads corresponding to the stores performed by save.

assembly language	restore Rd,Re,gh[Rf]
instruction format	EXP
opcode	E, 0B
machine language	E d 0B, e f gh
effect	$R_d := M[R_f + gh]$
	$\$R_{d+1} := M[R_f+gh+1]$
	$R_{d+2} := M[R_f + gh + 2]$
	...
	$R_e := M[R_f + gh + e - d]$

For example, consider this instruction:

```
restore  R3,R10,4[R14]
```

The effect is equivalent to

```
load  R3,4[R14]
load  R4,5[R14]
load  R5,6[R14]
load  R6,7[R14]
load  R7,8[R14]
load  R8,9[R14]
load  R9,10[R14]
load  R10,11[R14]
```

Suppose a stack frame was created by the `save` example above. To restore the registers from the stack frame, use:

```
restore R3,R9,6[R13]  ; e30b 9d06. store R3-R9 starting at
   6[R13]
```

It is equivalent to a sequence of store instructions:

```
load   R3,6[R13]
load   R4,7[R13]
load   R5,8[R13]
load   R6,9[R13]
load   R7,10[R13]
load   R8,11[R13]
load   R9,12[R13]
```

```
restore R3,R5,6[R13]  ; e30c 5d06. load R3-R5 starting at
   6[R13]
```

This is equivalent to a sequence of load instructions:

```
load    R3,6[R13]
load    R4,7[R13]
load    R5,8[R13]
load    R6,9[R13]
```

The **restore** instruction copies a sequence of consecutive memory loca-

tions starting from the effecive address into a sequence of adjacent registers. The index register (R14 in this example) is not changed. Restore is equivalent to a fixed sequence of load instructions; its purpose of restore is to restore the state of registers from memory after a procedure call or a context switch.

The instruction **restore Re,Rf,gh[Rd]** copies the contents of memory at consecutive locations beginning with mem[gh+Rf] into registers Re, Re+1, . . . , Rf.

5.4 Jumps

A jump is a transfer control to another address, rather than to the following instruction. The destination address is specified as the effective address in the jump instruction. A jump can be used to implement a goto statement.

5.4.1 jump

instruction format	RX
machine language	{$f d a $3} {disp}
effect	pc := disp+R[a]
assembly language	jump disp[Ra]

5.4.2 jumpc0, jumpc1

These instructions jump to a destination if a specified bit in the condition code is 0 (for jumpc0) or 1 (for jumpc1).

The first operand of jumpc0 is the index of a bit in R15. Don't specify R15; the condition bit is always in R15. The index of the condition bit must be specified in the d field of the instruction. Since this is an index, not a register, it's written as a number.

- jumpc0 3,loop[R0] ; Right
- jumpc0 R3,loop[R0] ; Wrong - not using R3!

```
lea     R1,5[R0]
lea     R2,6[R0]
cmp     R1,R2
jumpc0  0,yes[R0]  ; jump if GT so don't jump
jumpc0  4,no[R0]   ; jump if LT so do jump
```

jumpc0 goes to the destination if the specified condition bit is 0.

assembly language	jumpc0 d,disp[Ra]
instruction format	RX
opcode	f,4
machine language	$f d a $4
	disp
effect	if R[15].d=0
	then pc := disp+R[a]

jumpc1 is similar: it jumps if the specified bit is 1.

assembly language	jumpc1 d,disp[Ra]
instruction format	RX
opcode	f,5
machine language	$f d a $5
	disp
effect	if R15.d=1
effect	then R15.d=1

5.4.3 jal

assembly language	jal Rd,disp[Ra]
instruction format	EXP
opcode	f,6
machine language	{$f d a $6} {disp}
effect	R[d]:=pc, pc:=disp+R[a]

5.4.4 jumpz, jumpnz

The jumpz instruction jumps to the destination if a specified register is 0 (i.e. all the bits in that register are 0).

assembly language	jumpz d,disp[Ra]
instruction format	EXP
opcode	f,7
machine language	{$f d a $7} {disp}
effect	R[15].d=0 -> pc := disp+R[a]

The jumpnz instruction jumps to the destination if a specified register is not 0 (i.e. at least one of the bits in that register is 1).

assembly language	jumpnz d,disp[Ra]
instruction format	EXP
opcode	f,8
machine language	{$f d a $8} {disp}
effect	R[15].d=1 `> pc :` disp+R[a]

5.5 Logic

Sigma16 provides two logic instructions that can evaluate arbitrary Boolean expressions. They implement all possible logic functions, not just the most common ones. You can perform logic on individual bits, on a field within words, or on all the bits in a word.

- `logicf` performs bitwise logic on corresponding fields within two operand registers; the result overwrites the field in the first operand.
- `logicb` performs logic on two operand bits that may be at arbitrary indices within arbitrary registers. The result overwrites the first operand bit.

The assembly language provides a group of pseudoinstructions for the most common operations. Some of these (invw, andw, orw, xorw) are introduced in the tutorial section on logic. It is often more convenient and readable to use the pseudoinstructions, rather than the general `logicf` and `logicb`.

When translating a high level language program into assembly language, there are two ways to implement a Boolean expression: using logic instructions, or using a sequence of conditional jumps ("short circuit evaluation"). It's good practice to understand both methods and to use whichever is more appropriate in a specific situation. Just because an AND or OR operator

appears in a program, that doesn't necessarily mean that a logic instruction should be used. The logic instructions are important, and have many applications in practical programming, but it is sometimes better to use "short circuit evaluation" instead.

The next section explains how logic functions are represented, and the following sections describe each logic instruction followed by its pseudoinstructions.

5.5.1 General logic functions

The commonest Boolean operators are `inv`, `and2`, `or2`, and `xor2`. The `inv` function (also called invert, not, logical negation) has one input and produces one output, as defined by the truth table:

x	inv x
0	1
1	0

The `and2`, `or2`, `xor2` functions have two inputs, so their truth tables have four lines. There are also functions with more inputs, such as `and3` with 3 inputs, `and4` with 4 inputs, and so on. Sigma16 doesn't have instructions for those, but you can replace them with multiple uses of `and2`, `or2`, etc.

x	y	x and2 y	x or2 y	x xor2 y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

These aren't the only useful operations: there are 16 logic functions that take two arguments. Instead of providing specialised instructions for a few of the logic functions, Sigma16 provides general instructions that implement all of them.

Every logic function with two inputs x and y can be defined by a truth table where we list all possible values for the inputs and give the corresponding result. Each input could be either 0 or 1. Since there are 2 inputs, there are $2^2 = 4$ lines in the truth table.

The tables above for and2, or2, xor2 give specific values, either 0 or 1, for each result. For example, the result column for the `and2` function is (reading

from top to bottom) 0001, while `xor2` is 0110. For an arbitrary function, we can write the result column as *abcd* where each variable is either 0 or 1.

x	y	result
0	0	a
0	1	b
1	0	c
1	1	d

If $(a, b, c, d) = (0, 0, 0, 1)$ this is the truth table for `and2`. if $(a, b, c, d) = (0, 1, 1, 0)$ it is the truth table for `xor2`. Since there are 4 variables, there are $2^4 = 16$ possible settings of (a, b, c, d) and consequently there are 16 Boolean logic functions that take two arguments.

The Sigma16 logic instructions provide all 16 of these functions; taking a 4-bit code that specifies the logic function by giving the values of (a, b, c, d). In assembly language, the code is given as the decimal number represented by the bits (a, b, c, d), which is between 0 and 15. The codes for a few of the functions are:

abcd	code	name
0001	1	x and2 y
0111	7	x or2 y
0110	6	x xor2 y
1110	14	x nand2 y
1000	8	x nor2 y

The `inv` function can be treated as a 2-input function where the second argument is ignored.

Instead of writing the numeric code, we can define symbolic names for the logic functions we need:

```
and  equ    1
or   equ    7
xor  equ    6
```

Now, instead of specifying a logic function as (for example) 6, we can write it as `xor`.

5.5.2 Logic on fields: logicf

The logicf instruction performs a bitwise logic operation on fields within two words. The fields may be in different registers, but each field must be at the same position: their start and end indices must be the same. Each bit of the result is obtained by performing a specified logic function on the corresponding bits of the two fields. The result overwrites the field in the first operand register. The function h is specified as described above; for example, 6 is the code for exclusive or.

The field is specified as a pair of 4-bit constants f, g, where f is the index of the rightmost bit in the field and g is the index of the leftmost bit. Since the rightmost bit has a lower index, g should be greater than f. If $f = g$ then the field consists of just one bit. If $g < f$ then the field is empty, and the instruction will not change the destination register. The notation Rd.f:g means bits f through g in register Rd.

```
;    logicf  Rd,Re,f,g,h
;      Rd = first operand register and destination register
;      Re = second operand register
;      f = index of rightmost bit in field (in both registers))
;      g = index of leftmost bit in field (in both registers)
;      h = logic function
;      effect:  Rd.f:g := map2 h (Rd.f:g, Re.f:g)
;      format: EXP
;      machine language: Ed00 efgh (op=$E, ab=$00)
     logicf  R3,R9,5,8,xor   ; e300 9586
```

The following examples use the definitions of and, or, xor given above.

```
lea     R1,\$3333[R0]   ; R1 := 3333
lea     R2,\$5555[R0]   ; R2 := 5555
logicf  R1,R2,0,3,and  ; R1 := 3331
logicf  R1,R2,4,7,or   ; R1 := 3371
logicf  R1,R2,8,11,xor ; R1 := 3671
```

5.5.3 Word logic pseudoinstructions: invw, andw, orw, xorw

Several special cases of `logicf` appear frequently. There are special pseudoinstructions that simplify these special cases by allowing some of the operands

to be omitted.

The first simplification is to omit the operand that specifies the logic function, and instead to indicate the function in the mnemonic. For example, suppose you want to calculate the logical *and* of two fields. This can be done with the general logicf instruction:

```
logicf Rd,Re,f,g,1
```

Alternatively, you can indicate the logic function in the mnemonic and omit the function operand:

```
andf Rd,Re,f,g
```

This simplification can be used with the following four functions (the comment shows the actual machine instruction generated by the assembler):

```
;    invf   Rd,f,g        ; logicf Rd,R0,f,g,12
;    andf   Rd,Re,f,g     ; logicf Rd,Re,f,g,1
;    orf    Rd,Re,f,g     ; logicf Rd,Re,f,g,2
;    xorf   Rd,Re,f,g     ; logicf Rd,Re,f,g,6
```

The second simplification is to perform logic on an entire word, rather than a field within the word. This is achieved by specifying $f = 0$ and $g = 15$. The instruction `logicf Rd,Re,0,15,h` applies h bitwise to the entire regisers Rd and Re, and the result overwrites Rd. There are pseudoinstructions to simplify this (which were introduced in a tutorial section above):

```
invw  R3      ; R3 := invert R3
andw  R3,R4   ; R3 := R3 and R4
orw   R3,R4   ; R3 := R3 or R4
xorw  R3,R4   ; R3 := R3 xor R4
```

A pseudoinstruction is not a machine instruction: the Sigma16 processor does not have an `andw` instruction. Instead, the assembler generates the `logicf` instruction corresponding to any of `invw`, `andw`, `orw`, `xorw`. The following assembly language statements generate exactly the same machine language:

```
andw    R5,R7          ; R5 := R5 and R7
logicf  R5,R7,0,15,1   ; R5 := R5 and R7
```

If you're using one of the most common logic functions, a pseudoinstruction is convenient and can make a program more readable. If you prefer to use the actual machine instruction, rather than a pseudoinstruction, you can define a symbol for the logic function using an `equ` assembly directive. The symbol can then be used instead of the numeric code:

```
xor  equ   6
     \ldots{}
     logicf R5,R7,R2,xor   ; R5 := R7 xor R2
```

5.5.4 Bit logic: logicb

The bit logic instruction `logicb` specifies two operand bits, which may be in any register at any index. It applies a specified logic function to these bits and puts the result bit back into the first operand, overwriting the previous value of that bit. The other bits in the first operand register remain unchanged. As this instruction modifies only one bit in the first operand, it enables you to keep many Boolean variables in just one register.

The general form is:

```
;    logicb  Rd,Re,f,g,h
;      effect: Rd.f := h (Rd.f, Re.g)
;      format: EXP
;      machine language: Ed01 efgh (op=E ab=01)
     logicb  R7,R9,3,5,6    ; e701 9356
```

Suppose you want to calculate the logical xor of the bit in R3 at index 12 and the bit in R7 at index 14. This instruction will do it (provided that the symbol xor is defined to be 6), The result will overwrite the bit in R3 at index 12:

```
logicb  R3,R7,12,14,xor  ; R3.12 := R3.12 xor R7.14
```

Example:

```
lea    R2,\$0f00[R0]   ; R2.9 := 1
lea    R7,\$2000[R0]   ; R7.13 := 1
logicb R2,R7,9,13,xor ; R2 := 0b00 (R2.9 := R2.9 xor R7.13 = 0)
```

5.5.5 Pseudoinstructions invb, andb, orb, xorb

There are pseudoinstructions for the commonest logic functions:

```
invb    R1,3,R2,9   ; R1.3 := inv R2.9
andb    R1,3,R2,9   ; R1.3 := R1.3 and R2.9
orb     R1,3,R2,9   ; R1.3 := R1.3 or R2.9
xorb    R1,3,R2,9   ; R1.3 := R1.3 xor R2.9
```

The `logicf` instruction and its pseudoinstructions (`andf`, etc.) can operate on fields of any size, including 1-bit fields. However, `logicb` and its pseudoinstructions are more flexible than `logicf` on a 1-bit field, because `logicb` allows the operands to have different bit indices.

5.5.6 Pseudoinstructions setb, clearb, copyb, copybi

The bit logic instructions have some useful special cases that don't appear to involve logic at all. These are supported by pseudoinstructions.

- `setb Rd,f` puts 1 into a specified bit: `Rd.f : 1=`.
- `clearb Rd,f` puts 0 into the specified bit.
- `copyb Rd,Re,f,g` copies a bit from register Re at index g into register Rd at index f.
- `copybi Rd,Re,f,g` is similar to copyb, except that it inverts the bit as it is written into the destination.

Here are some examples:

```
setb   R4,7         ; R4.7 := 1
clearb R4,7         ; R4.7 := 0
copyb  R4,R12,7,5   ; R4.7 := R12.5
copybi R4,R12,7,5   ; R4.7 := inv R12.5
```

5.6 Bit manipulation

5.6.1 Shifting: shiftl, shiftr

The shift instructions treat the operand as a string of bits, and move each bit a fixed distance to the left or right.

The instruction shiftl `Rd,Re,h` shifts the value in the operand register Ra by h bits to the left, and the result is placed in the destination register Rd. The operand Re is not modified. The leftmost h bits of the operand are discarded and the rightmost h bits of the result become 0. The general form is

```
;    shiftl Rd,Re,h
;      effect:  Rd := Re shl h
;      Rd = first operand register and destination register
;      Re = second operand register
;      f = unused
;      g = unused
;      h = constant shift distance
;      format: EXP/RRk
;      machine language: Ed04 efgh (op=$E, ab=$03)
     lea    R1,2[R0]   ;           R2 = 0002  0000 0000 0000 0010
     shiftl R5,R1,4    ; E503 1004 R5 = 0020  0000 0000 0010 0000
```

The instruction `shiftr Rd,Re,h` shifts the value in the operand register Re by h bits to the right, and the result is placed in the destination register Rd. The operand Re is not modified. During the shift, the rightmost k bits of the value are discarded and the leftmost k bits become 0. The general form is

```
;    shiftr Rd,Re,h
;      effect:  Rd := Re shr h
;      Rd = first operand register and destination register
;      Re = second operand register
;      f = unused
;      g = unused
;      h = constant shift distance
;      format: EXP/RRk
;      machine language: Ed04 efgh (op=$E, ab=$04)
```

The following instruction shifts the value in R3 to the right by 5 bits and place the result in R2. The operand register R3 is not changed.

```
shiftr  R2,R3,5
```

5.6.2 Bit fields: extract

The extract instruction copies a field within a register to another register. The extract instruction copies an arbitrary field of bits from a source register and inserts them into an arbitrary position in a destination register. The destination field is overwritten, while other bits in the destination register as well as all bits in the source register are unchanged.

These instructions are useful for systems programming. Emulators need to access instruction fields, software implementing floating point needs to access the parts of a floating point number, and networking software needs to decode message headers.

Bit indexing. Sigma16 indexes bit positions in a word from right to left, starting from 0. The least significant (rightmost) bit has index 0, and the most significant (leftmost) bit has index 15.

A bit field is a sequence of bits within a word. A field is specified using a pair of 4-bit natural numbers L, R, where L is the index of the leftmost bit in the field and R is the index of the rightmost bit. The size of a field is $\max(0, L - R + 1)$. Consequently, if $L < R$ the size is 0. For example:

- R8.6,4 is the field consisting of R8.6, R8.5, R8.4 and its size is 3.
- R8.11,11 is the field consisting of R8.11 and its size is 1.
- R8.15,0 is the field containing the entire contents of R8 and its size is 16.
- R8.5,7 is an empty field containing no bits; its size is 0

Machine language. The machine language format of extract is EXP, with the following fields:

op	d	a	b
e	f	g	h

- op = $e (escape to EXP)
- Rd = destination register
- ab = secondary opcode: $05 for extract
- Re = source register
- f = destination start index L
- g = destination end index R
- h = source start index L

The instruction specifies the destination field as f, g. The field size S and the R index for the source are not specified in the instruction: only the leftmost index of the source field is specified explicitly as h.

The sizes of the source and destination fields must be the same. The size and right index of the source are calculated as follows. The calculations verify that both the source and destination fields have the same size.

- The size S of a field specified by L, R is $\max(0, L - R + 1)$.
- In general, $R = L - S + 1$.
- Destination field = Rd.f, Rd.f-1, ..., Rd.g.
- Check that this works for destination: R = L - S + 1 = f - (f-g+1) + 1 = f - f + g - 1 + 1 = g.
- Source field = Re.h, Re.h-1, ..., Re.h-f+g
- Calculate R for source, which is not specified in the instruction. R = L - S + 1 = h - (f-g+1) + 1 = h - f + g - 1 + 1 = h - f + g.

Assembly language. The assembly language operand format for extract is RkkRk, where R denotes a register number and k denotes a 4-bit constant. The general form of the instruction, in assembly language, is

```
extract Rd,f,g,Re,h
```

Effect. The effect is to overwrite each bit in the destination field with the corresponding bit in the source field:

```
Rd.f := Re.h
Rd.f-1 := Re.h-1
\ldots{}
Rd.g := Re.h-f+g
```

For example, consider `extract R14,9,7,R13,5`, where $f = 9$, $g = 7$, $h = 5$. The field size is $f-g+1 = 9-7+1 = 3$, so 3 bit assignments take place. The index of the rightmost bit in the source field $= h-f+g = 5-9+7 = 3$. Therefore the instruction performs the following bit assignments:

```
R14.9 := R13.5
R14.8 := R13.4
R14.7 := R13.3
```

```
; Assembly language: extract Rd,f,g,Re,h
; Effect:            Rd.f..g := Re.h..(h+g-f)
; EXP opcode:        e,15

; Example:           extract R2,11,8,R1,3
; Machine language:  e215 1b83
```

Example:

```
extract R2,7,4,R3,20
R2.7 := R3.20
R2.6 := R3.19
R2.5 := R3.18
R2.4 := R3.17
```

The extract instruction can be implemented using a combination of logic and shift instructions. It is included in the architecture for several reasons:

- extract provides useful abstractions for writing interpreters and simulators.

- When used in an interpreter, bit field operations are executed frequently: they are a crucial part of the "inner loop". Therefore the efficiency of common bit field operations is important.

- The bit field instructions are easier to use and more readable than the corresponding logic and shifts.

- This instruction can be implemented efficiently in a digital circuit and this implementation is an interesting design problem.

The effect of an extract instruction can be described by writing each bit assignment individually, so there are *size* individual bit assignments. The notation R2.7 means *the bit at position 7 in register R2*. Using this notation, the example above performs the following bit assignments:

extract R2,R1,11,3,4
R2.11 := R1.3
R2.10 := R1.2
R2.9 := R1.1
R2.8 := R1.0

Relation to other instructions. Although extract performs a number of bit assignments, it is a single instruction and its execution time is a small fixed number of clock cycles. The execution time does not depend on the value of the field size, and extracting a large field doesn't require more time than extracting a small field. The hardware implementation of the instruction does not use an iteration to copy the bits; they are all copied in parallel in one clock cycle.

Any register may be specified for the source and destination. If they are the same, the effect is to move a bit field from one place to another within the register (this is not the same as a shift). If the destination is R0, the result is discarded and the instruction has no effect.

These instructions can be implemented using a combination of logic and shift instructions. They are included in the architecture for several reasons:

- These operations provide useful abstractions for writing interpreters and simulators.

- When used in an interpreter, bit field operations are executed frequently: they are a crucial part of the "inner loop". Therefore the efficiency of common bit field operations is important.

- The bit field instructions are easier to use and more readable than the corresponding logic and shifts.
- These instructions can be implemented efficiently in a digital circuit and this implementation is an interesting design problem.

5.7 User requests

5.7.1 Request to OS: trap

5.7.2 testset

assembly language	testset Rd,disp[Ra]
instruction format	EXP
opcode	f,9
machine language	{$f d a $9} {disp}
effect	R[d]:=M[disp+R[a], M[disp+R[a]]:=0001

5.8 System control

5.8.1 Accessing control registers: getctl, putctl

5.8.2 Context switching: resume

5.8.3 Timer: timeron, timeroff

6 Summary of instruction set

- The Mnemonic gives the symbolic name of the instruction used in assembly language
- The ISA specifies the smallest subset of the instruction set architecture that allows this instruction. The Core subset is a minimal architecture; the Standard subset contains a full range of instructions
- The R15 entry indicates how the instruction uses R15: cc indicates the instruction sets the condition code in R15, and x indicates the instruction places data in R15. A blank entry means the instruction does not change R15.
- The P entry indicates whether the instruction is Privileged (P) or not (blank)

The Args column shows the assembly language statement format.

- R is a register, e.g. R4 or R15
- D is an expression denoting the displacement, which must be representable in 16 bits
- k is a small constant representable in 4 bits, so 0 <= k < 16
- K is a large constant representable in 12 bits, so 0 <=k < 4096

The first word of an instruction:

Op	D J	A	B

Fields of the first instruction word:

- Op = bits 15..12. Used in all instructions as the primary operation code
- D = bits 11,10,9,8. Destination register (e.g. R5)
- A = bits 7..4. First operand register (e.g. R12)
- B = bits 3..0. is either second operand register (e.g. R7) or secondary opcode (4-bit constant)
- k is a 4-bit constant in the d field, (e.g. 5)
- ab is 8-bit secondary opcode for EXP format (e.g. $c3)
- C is a control register specified by 4 bits
- A "don't care" field is indicated with a dash

The second word of an instruction, for RX and EXP formats:
Formats for the second word of an instruction. The second word may consist of one of the following:

- disp - a 16-bit constant

- kq - a 4-bit constant k (in the e field), followed by a 12-bit constant q (in the fgh fields)

The following table summarises the instructions in the Core subset of Sigma16. The columns are:

- Mnemonic. The assembly language name of the instruction
- ISA. The Instruction Set Architecture subset that contains the instruction; for this table all the instructions are Core.
- P. * indicates that the instruction is privileged, blank indicates that it is not. All of the Core instructions are unprivileged. For the meaning of "privileged", see the System section.
- Fmt. The instruction format. There are two Core instruction formats: RRR (instruction has three operand fields, each a register) and RX (instruction has a register operand and an X operand consisting of a displacement constant and an index register).
- Args. The assembly language argument format. Usually this is the same as Fmt. However, some instructions don't use all the fields, and the assembly language statement omits the irrelevant field (e.g. cmp is RRR format but the assembly language statement omits the d field, which is ignored).
- Code.
- Effect. A statement in an imperative programming language which describes what the instruction does.

Assembly directives

- data

6.1 RRR instructions

Assembly	Op	Code	Effect
add Rd,Ra,Rb	0	$0 d a b	R[d] := R[a] + R[g]
			R15 ~= gGElvVC
sub Rd,Ra,Rb	1	$1 d a b	R[d] := R[a] - R[b]
mul d,a,b	2	$2 d a b	R[d] := R[a] * R[b]
div d,a,b	3	$3 d a b	R[d] := R[a] / R[b]
			R[15] := R[a] % R[b]
cmp d,a	4	$4 - a b	R[15] := R[a] cp R[b]
addc d,a,b	5	$5 d a b	Rd := Ra + Rb + R15.C)
muln d,a,b	6	$6 d a b	Rd := Ra :*: Rb
			R15 :=
divn d,a,b	7	$7 d a b	R15++Rd := R15++Ra
			:/: Rb
	8	$8 - - -	nop
	9	$9 - - -	nop
	a	$a - - -	nop
	b	$b - - -	nop
trap d,a,b	c	$c d a b	interrupt
	d	$d d a b	escape reserved
	e	$e d ab	escape EXP op=ab
	f	$f d a b	escape RX op=b

6.2 RX instructions

Assembly	Op	Code	Effect
lea Rd,disp[Ra]	f,0	$f d a $0 disp	R[d] := disp+R[a]
load Rd,disp[Ra]	f,1	$f d a $1 disp	R[d] := M[disp+R[a]]
store Rd,disp[Ra]	f,2	$f d a $2 disp	M[disp+R[a]] := R[d]
jump disp[Ra]	f,3	$f d a $3 disp	pc := disp+R[a]
jumpc0 d,disp[Ra]	f,4	$f d a $4 disp	if R[15].d=0 then pc := disp+R[a]
jumpc1 d,disp[Ra]	f,5	$f d a $5 disp	if R[15].d=1 then pc := disp+R[a]
jal Rd,disp[Ra]	f,6	$f d a $6 disp	R[d]:=pc, pc:=disp+R[a]
jumpz Rd,disp[Ra]	f,7	$f d a $7 disp	if R[15].d=0 then pc := disp+R[a]
jumpnz Rd,disp[Ra]	f,8	$f d a $8 disp	if R[15].d=1 then pc := disp+R[a]
testset Rd,disp[Ra]	f,9	$f d a $9 disp	R[d]:=M[disp+R[a]], M[disp+R[a]]:=0001
	f,a	$f d a $a disp	nop, reserved
	...		...
	f,f	$f d a $f disp	nop, reserved

6.3 EXP instructions

Instructions and pseudoinstructions in each format:

- EXP/: resume
- EXP/R pseudo: invf
- EXP/Rk: dispatch (k is efgh)
- EXP/RR pseudo

  - andw, orw, xorw, andf, orf, xorf
  - Rd = first operand register and destination register
  - Re = second operand register
  - f = unused
  - g = unused
  - h = generated by pseudo expansion
- EXP/RRk: shiftl, shiftr
- EXP/RRkR: save, restore
- EXP/RRkk pseudo: andb, orb, xorb, invb, orb, xorb
- EXP/RRkkk: logicf, logicb, extract

119

Assembly	Op	Code	Effect
logicf Rd,Re,f,g,h	e,00	Ed00 efgh	Rd.f:g := h#(Rd.f:g,Re.f:g)
logicb Rd,Re,f,g,h	e,01	Ed01 efgh	Rd.f := h(Rd.f,Re.g)
shiftl Rd,Re,h	e,03		Rd := Re shl h
shiftr Rd,Re,h	e,04		Rd := Re shr h
extract Rd,Re,f,g,h	e,05		Rd.f~g :=
push Rd,Re,Rf	e,07		M[ea] := Rd, Rb++
pop Rd,Re,Rf	e,08		Rb–, Rd := M[ea]
top Rd,Re,Rf	e,09		Rb–, Rd := M[ea]
save Rd,Re,gh[Rf]	e,0a		M[Rf+gh] := Rd M[Rf+gh+1] := Rd+1 ... M[Rf+gh+e-d] := Re
restore Rd,Re,gh[Rf]	e,0b		Rd := M[Rf+gh] Rd+1 := M[Rf+gh+1] ... Re := M[Rf+gh+e-d]
brc0 Rd,e,fgh	e,0c	Ed0C efgh	if R[d].e=0 then pc:=pc+fgh
brc1 Rd,e,fgh	e,0d e,0d	Ed0D efgh	if R[d].e=1 then pc:=pc+fgh
brz Rd,efgh	e,0e	Ed0E efgh	if R[d]=0000 then pc:=pc+efgh
brnz Rd,efgh	e,0f	Ed0F efgh	if R[d]!=0000 then pc:=pc+efgh
dispatch Rd,efgh	e,10	Ed10 efgh	pc := M[pc+min(Rd,efgh)]]
getctl Rd,c	e,11		Rd := Sc privileged
putctl Rd,c	e,12		Sc := Rd privileged
resume	e,13		pc := ipc, (privileged) status := istatus
timon Rd,efgh	e,14		timer := efgh, start timer
timoff Rd,efgh	e,15		stop timer
	e,16		reserved: nop

6.4 Sigma32 EXP

120

Assembly Op Code Effect

ladd Rd,Re,Rf	e,16	Ed16	Rd := Re+rf
		ef00	Rd := Re+rf
lsub Rd,Re,Rf	e,17	Ed17	Rd := Re+rf
		ef00	Rd := Re+rf
lmul Rd,Ra,Rb	e,16	Ed16	Rd := Re+rf
ldiv Rd,Ra,Rb	e,16	Ed16	Rd := Re+rf
lcmp Rd,Ra,Rb	e,16	Ed16	Rd := Re+rf
llea Rd,disp[Ra]	f,0		R[d] := disp+R[a]
lload Rd,disp[Ra]	f,1		R[d] := M[disp+R[a]]
lstore Rd,disp[Ra]	f,2		M[disp+R[a]] := R[d]
llogicf Rd,Re,f,g,h	e,00	Ed00	Rd.f:g := h#(Rd.f:g,Re.f:g)
llogicb Rd,Re,f,g,h	e,01	Ed01	Rd.f := h(Rd.f,Re.g)
lshiftl Rd,Re,h	e,03		Rd := Re shl h
lshiftr Rd,Re,h	e,04		Rd := Re shr h
lextract Rd,Re,f,g,h	e,05		Rd.f$\tilde{g}$:=

6.5 Summary of pseudoinstructions

Pseudoinstructions for comparisons

- jumplt jump if <
- jumple jump if <=
- jumpeq jump if =
- jumpne jump if !=
- jumpge jump if >-
- jumpgt jump if >

Assembly	Op	Format	Code	Effect
invw Rd		pseudo R	Ed01 e0F1	logicf Rd,Re,0,15,1
invw Rd		pseudo R	Ed01 e0F1	logicf Rd,Re,0,15,1
andw Rd,Re		pseudo RR	Ed01 e0F1	logicf Rd,Re,0,15,1
orw Rd,Re		pseudo RR	Ed01 e0F1	logicf Rd,Re,0,15,1
xorw Rd,Re		pseudo RR	Ed01 e0F1	logicf Rd,Re,0,15,1
invf Rd,f,g		pseudo R	Ed01 e0F1	logicf Rd,Re,0,15,1
andf Rd,Re,f,g		pseudo RR	Ed01 e0F1	logicf Rd,Re,0,15,1
orf Rd,Re,f,g		pseudo RR	Ed01 e0F1	logicf Rd,Re,0,15,1
xorf Rd,Re,f,g		pseudo RR	Ed01 e0F1	logicf Rd,Re,0,15,1
invb Rd,Re,f,g		pseudo RRkk	Ed01 efg1	logicb Rd,Re,f,g,1
andb Rd,Re,f,g		pseudo RRkk	Ed01 efg1	logicb Rd,Re,f,g,1
orb Rd,Re,f,g		pseudo RRkk	Ed01 efg1	logicb Rd,Re,f,g,1
xorb Rd,Re,f,g		pseudo RRkk	Ed01 efg1	logicb Rd,Re,f,g,1

7 Assembly language

:CUSTOM$_{\text{ID}}$: sec-assembly-language

A computer is a digital circuit that executes programs in machine language, which is hard for humans to read because it consists entirely of numbers. Assembly language provides a readable notation for writing machine language programs. It uses names for instructions and variables, as well as other notations to make the code easier to understand.

An instruction in machine language is just one or more words (often

written in hexadecimal notation), while the corresponding instruction in assembly language uses mnemonic names so the programmer doesn't have to memorise all the operation codes, addresses of variables, and so on. For example, the assembly language statement mul R12,R3,R8 is more readable than the corresponding machine language instruction 2c38. However, the assembly language still gives the programmer complete control over every bit a program.

A programmer writes a machine-level program in assembly language. A software application called the **assembler** reads it in and translates it to machine language. When it sees an instruction mnemonic like add or div, it replaces it with the operation code (0, 3, or whatever). The assembler helps with variable names — the machine language needs addresses (numbers) and the assembler calculates them

- You can use names (add, div) rather than numeric codes (0, 3)
- You can use variable names (x, y, sum) rather than memory addresses (02c3, 18d2)
- You write a program in assemply language
- The assembler translates it into machine language

Compilers and assemblers are similar in some ways: both of them translate a program from one language to another. The main difference is that compilers translate between languages that are quite different, while assemblers translate between similar languages.

Example: a sequence of RRR instructions

Assembly language

```
add    R3,R5,R1
sub    R4,R2,R3
mul    R1,R9,R10
```

Machine language

```
0351
1423
219a
```

7.1 Programs, modules, and files

Sigma16 has the flexibility required for "programming in the large". It provides modules, separate assembly, import and export, relocation, linking, executables, and booting. It also has special conventions that enable the user to skip those complications, and simply enter a standalone program, assemble it, and run it. The system is straightforward for beginners but allows a transition to realistic systems programming.

The assembler inputs a program in assembly language. This is called the *source module*. The assembler outputs an *object module* that contains the machine language code. The assembler also produces an *assembly listing*, which presents the program in a form useful for the programmer. The assembly listing shows the source code, the corresponding machine language, a symbol table, and any error messages. Finally, the assembler outputs a *metadata module* that enables the emulator to track the source statement corresponding to each instruction.

A simple program consists of just one source module that does not import anything. This is a *standalone program*, and assembling it produces an *executable program*. An executable program is an object module that does not require linking; it can be booted directly in the processor.

A larger program may consist of several source modules, including one main program. This requires use of the *linker* to combine the collection of object modules into a single executable program.

7.1.1 Standalone programs

If a program consists of just one source module that does not import any names, it is *standalone*. There are several ways to input a standalone program: you can choose it from one of the examples, load it from a file, or type it into the editor. In all cases, the text of the program is shown in the editor pane. Go to the Assembler pane and click Assemble. If there are no errors, go directly to the Processor pane (you can skip the Linker) and click Boot. This will read the machine language into the memory, and now you can run the program.

7.1.2 Modules

Large programs are easier to develop if you break them into separate modules which can be assembled separately. One of the modules is identified as the

main module, and it imports the other modules. This allows you to define procedures, global variables, and symbolic constants in separate modules. Many separate programs can reuse these modules simply by importing them.

The Sigma16 app maintains a *Module Set* describing all the modules that it knows about. The usual workflow is to read in all the modules comprising a program, and to assemble each of them. Then the linker combines all the modules in the Module Set to form an executable. The executable can be saved to a file, or it can be booted into the processor.

The Module Set is displayed in the *Modules* page. Key information about each module is shown, including the first few lines of the source code. It is good practice to begin every source module with two or three lines of comments that identify the code, giving its name, purpose, author, and date.

The app maintains several invariants:

- At all times, there is at least one module. If you close the last remaining module, a new one with empty source code is immediately created, to ensure that there is at least one module.
- At all times, exactly one module is *selected*. The assembler always works on the selected module. When you go to the Assemble page, the assembler source code is set to the text of the selected module.
- At all times, the source code text of the selected module is displayed in the editor window.

Each module in the Module Set has a *module record* that contains everything the app knows about the module:

- (optional) the name of the module
- (always) the source code (which could just be an empty or blank string)
- (optional) the filename and file handle of the source code
- (if there is a file handle) whether the code in the file is *stale* (i.e. the current source code has been edited and differs from the text in the file)
- (after assembly) the object code. Each module has an associated object code, which may be empty. The object code can be produced by

a successful assembly (i.e. an assembly with no errors) or it can be obtained from the Editor. This allows object code to be read from a file or entered directly by the user.

- (after assembly) the assembly listing
- (after assembly) the metadata

1. Creating a module

    - *Launch.* When the Sigma16 app is launched, it creates the initial Module Set containing a module whose source text is the empty string. This module is automatically selected, and it is not associated with a file. As always, you can see the source code text (which is empty) by going to the Editor.
    - *Load example.* You can open one of the Examples: Go to the Examples page, navigate through the index pages to one of the programs, and select it. This will add a new module to the Module Set, select it, and set its source code to the contents of the file. The object code is set to empty.
    - *Read file.* You can read a source file from your user space to create a module. Go to the Modules page and click Choose Module. This adds the result to the Module Set. The object code is set to empty.
    - *Editor: New.* Go to the Editor page and click *New*. This creates a new module whose name is `Anonymous` whose source text is the empty string. The new module is selected and added to the Module Set (you can see it by going to the Modules page). The previous text in the editor page is not lost; it remains in the module that had previous been selected.

2. Selecting a module

    The Modules page shows which module is currently selected, and its source text is visible if you go to the Editor page.

3. Changing a module

    - *Edit text.*
    - *Save.*

- *Select.*

4. Closing a module
    - *Close.*

7.1.3 Modules page

- In Modules tab, click Choose Files
    - The dialogue shows .asm.txt, .lnk.txt, .obj.txt, .md.txt
    - To select all the relevant files in a directory, click the little box at the left on the row giving "Name, date modified, . . . "
    - Click Open (or cancel)

7.1.4 Editor page

Editor operations on files and modules

Each operation that changes the editor buffer (New, Open, Close) checks first to see whehter the buffer has been changed since it was last saved. If so, a dialogue asks whether the file should be saved.

- New – Check whether text in the editor buffer has been saved; if not, ask whether to save it. Create a new module with empty text and no file name, add it to the module set, and select it as the current module. Clear the text in the editor buffer.

- Open – Check whether text in the editor buffer has been saved; if not, ask whether to save it. Enter the open file dialogue where an existing file can be found by navigation or by typing in its name. If the dialogue is cancelled, the module set and editor buffer are left unchanged. If a file is selected in the dialogue, and it is already in the module set, then it is selected in the current module. Otherwise, a new module is created with the file's contents, and is selected as the current module. and The file is loaded into the editor buffer and added to the module set.

- Refresh – The file corresponding to the current module is read, and its contents are loaded into the editor buffer.

- SaveAs – Enters the save file dialogue where the directory and file name can be chosen. The editor buffer is written into this file.
- Save – Writes the editor buffer into the current file and directory. If either the module name or directory is not known, this reverts to a SaveAs.
- Select – Opens a list of all modules; you can click one of them which is then set as the current module.
- Close – Check whether text in the editor buffer has been saved; if not, ask whether to save it. The module is removed from the module set, and the editor buffer is cleared. The first module (module number 0) is selected as the current module, but if there is no module at all, an empty module is created and selected as current (in effect, if there is only one module and you close it, an automatic New is performed).
- Example – Reads in a simple example program and sets it as the current module. This is a standalone program; you can simply click Editor: Example, then Assembler: Assemble, then Processor: Boot, and run the program. This example is used in the first "getting started" tutorial. The example program is also available in the Examples directory, accessible through Editor: Open.

Select is for switching among the existing modules, while New and Open are for introducing a new module.

7.1.5 Files

A source module may be stored in a file, although this is not required. The assembler will produce several objects, which can also be stored into files.

There is a standard convention for Sigma16 filenames. All filenames end with a two-part extension, such as `.asm.txt`. The first part (`.asm`) tells Sigma16 what kind of information the file contains, and the final part (`.txt`) enables the computer you're running on to interpret the file as plain text. If you have a program named `MyProgram`, then the files associated with it must be named as follows:

Description	Language	Filename
source code	assembly language	MyProgram.asm.txt
object code	object language	MyProgram.obj.txt
assembly listing	plain text	MyProgram.lst.txt
metadata	plain text	MyProgram.md.txt

The first part of the filename (`MyProgram`) is the *base name*, and is the same for all the files for the module. The last part (`.asm.txt`) is the *extension*.

A filename can be specified either as a full path (the unique identification of the file (`C:\\Users\...\prog.asm.txt`), or as just a filename (myprogram.asm.txt) which is relative to the current directory.

To edit a file, the modDir and modName are both optional. An edited file may have a module name specified with a module statement. To read or save a file, both the module directory and name must be known.

There are two ways to read in a file:

- Go to the *Examples* page, navigate to one of the examples, and open it. This will create a new module record and read the `.asm.txt` file into the source code for that record. The object, assembly listing, and metadata for the module record are set to the empty string.
- Go to the *Modules* page, click *Choose File*, and navigate to a source file anywhere in your file space. Selet that file, and it will be read into the source code for a newly created module record.

7.2 Fixed and relocatable values

A value is a 16-bit word. An assembly language program uses expressions to denote values, but the actual underlying quantity is a value. A value consists of a word and several attributes:

- word is a natural number in the range from 0 to 2^{16}-1.
- origin
  - if origin=Local, the value is defined within the module
  - if origin=External, the value is imported from another module
- movability

- If movability = Relocatable, the value must be adjusted by the relocation constant when the module is relocated
- If movability = Fixed, the value is not affected during relocation

7.2.1 Expressions

An expression is syntax that denotes a value.

A **name** must begin with a letter (a-z or A-Z), and may contain letters, digits, or underscore characters.

Constants can be written in decimal, hexadecimal, or binary:

- **Decimal constants** consist of a sequence of digits, with an optional leading - sign. Examples: 42 55039 -1
- **Hexadecimal constants** are written with a dollar sign $ followed by four hex digits (0 1 2 3 4 5 6 7 8 9 a b c d e f). Examples: $0249 $c78a
- **Binary constants** are written with a hash sign # followed by any number of 0 or 1 characters. You can write fewer than 16 bits; they will be padded on the left with zeros. Examples: #1101 #000100000001101

Expressions may contain arithmetic operators + - * /.

operand	operator	operator	result
fixed	+	fixed	fixed
fixed	+	relocatable	relocatable
relocatable	+	fixed	relocatable
relocatable	+	relocatable	error
fixed	-	fixed	fixed
fixed	-	relocatable	relocatable
relocatable	-	fixed	relocatable
relocatable	-	relocatable	fixed
fixed	*	fixed	fixed
fixed	*	relocatable	error
relocatable	*	fixed	error
relocatable	*	relocatable	error
fixed	/	fixed	fixed
fixed	/	relocatable	error
relocatable	/	fixed	error
relocatable	/	relocatable	error

relocatable

An expression can do arithmetic on a local label, but not on an imported name. The reason is that arithmetic requires that the value of the name is known. That's why an expression like a equ rcd+5 can be used only after the label rcd is defined: it enables the value of each name to be calculated during pass 1. But the values of imported names are not known at all during assembly; they become defined only during linking. Such a value can affect the values of words in the object code, but not their locations.

An expression is assembly language syntax that, when evaluated, denotes a value (i.e. a 16-bit word). Evaluation takes place entirely at assembly time. Expressions may be labels, constants, or may be calculated.

7.2.2 Location counter

The assembler maintains a variable called the location counter, which is the address where the next word of object code will be loaded. The location counter is a local value. It is initialized to 0000 Relocatable.

When an instruction word or data word is generated, its address is set to the current value of the location counter, which is then incremented.

The org directive specifies a new value of the location counter. First the operand of the org statement is evaluated. This value must be local (it is an error if the value is external). The location counter and its movability are set to the value and movability of the operand.

7.2.3 Attributes

A machine language program consists of words stored in memory at particular addresses. A word is just a collection of 16 bits; it has no type.

An assembly language program specifies all the words that comprise a program. In principle you could just write out all the words as numbers, but this is difficult and prone to errors. Thw whole point of assembly language is to provide notations that make it easier to specify these numbers, while retaining total control—every single bit in the object code is determined by the assembly language.

A **value** is a word of 16 bits. Values do not have types; their type depends entirely on usage. Values may be used in generating object code, either as the displacment field of an RX instruction or as the operand of a data statement.

Every value is either **fixed** or **relocatable**. If a module is linked, then its relocatable values may be translated, but the fixed values remain unchanged.

7.3 Code generators

Each line of source code is an assembly language statement. Unlike higher level languages, assembly language statements are not nested. There are three kinds of assembly language statement:

- *Comments* (blank lines, or lines beginning with ;)
- *Code* statements define instructions or constant data
- /Directives provide metadata but don't generate any code

An assembly language statement contains one or more fields. A field consists of non-space characters (with one exception: a space may appear in a string literal). Fields are separated from each other by one or more white space characters.

- Label. The label field is optional. If present, the label must be a name and it must begin in the first character of the line. If the first character is a space, then that line has no label.
- Operation. The operation field is an identifier that specifies an instruction or assembler directive. It must be preceded by one or more white space characters. Every statement (apart from a full line comment) must have an operation field.
- Operands. The operands field specifies operands for an instruction or arguments for assembly directives. There may be several operands, which must be separated by commas. Each type of statement (determined by the operation field) requires a specific syntax for the operands. Most instructions and assembler directives require operands, but some do not.
- Comment. All text that either (1) follows white space after the operands field, or (2) follows a semicolon (;), is a comment, and is ignored by the assembler. If one or more of the other fields (label, operation, operands) is missing, the comment must be preceded by a semicolon to prevent it

from being interpreted as operands. The rule is: all text after a semicolon is a comment, and all text after white space following operands is a comment. A statement where the first non-space character is a semicolon is a full line comment. If the statement has no operands, then all text after the operation field is a comment. It is good practice always to begin a comment with a semicolon.

7.3.1 Instructions

Assembly language statements generally correspond to the instruction formats, but there is not an exact correspondence for several reasons:

- Sometimes an instruction is written in assembly language with a field omitted which exists in the machine language code but is ignored. For example, the instruction **cmp R1,R2** generates an RRR instruction, but the third operand field is omitted because the instruction requires only one operand, not two. The assembler sets the unused operand to 0, but the machine ignores it. This is called a "don't care" field in the instruction.

- Sometimes two instructions look the same in assembly language but use different machine language instruction formats. For example, **add R1,R2,R3** and **push R1,R2,R3** look similar, but **add** uses the RRR instruction format and **push** uses the EXP instruction format. The reason for this is that there are not enough bits in the op field to accommodate all the instructions with three register operands, so an **expanding opcode** is used. Thus push is represented with op=14, indicating EXP format, and the EXP variant is used for this instruction.

- The 4-bit fields are sometimes used to denote a register from the register file (R3), or a control register (mask), or a constant . In assembly language the constants are written just as a number (e.g. shiftl R1,R2,5). Control registers are written by name rather than their number in the control register file (e.g. getctl R3,mask).

- Some assembly language statements are **pseudoinstructions**. These are special cases of more general instructions. For example, **and** is a pseudoinstruction which generates a **logicf** instruction specialised to perform a logical and.

Table: **Assembly language statement formats**

afmt	operand general form	example	notes
RRR	Rd,Ra,Rb	add R1,R2,R3	
RR	Ra,Rb	cmp R1,R2	
RX	Rd,disp[Ra]	load R1,xyz[R2]	
kX	k,disp[Ra]	jumpc0 6,loop[R2]	0 <= k <= 15
RRRk	Rd,Ra,Rb,K[Re]	save R1,R5,3[R14]	0 <= K <= 255
RRk	Ra,Rb,k	shiftl R1,R2,5	0 <= k <= 15
Rkkkk	Rd,e,f,g,h		
RkkRk	Ra,j,n,Rb,k	extract	
RC	Rc,Rd	putctl vect,R4	

Label	Statement	Operands	Purpose
optname	data	exprs	generate word for each exp
	RRRop	r,r,r	
	RXop	r,exp[r]	
	RRop	r,r	
	Rop	r	
	RRKKop	r,r,k,k	

Asm	instruction	operands	ML formats
RRR	add	Rd,Ra,Rb	RRR
RX	lea	Rd,disp[Ra]	RX
RR	inv	Rd,Ra	RRR (b ignored), RREXP
JX	jump	disp[Ra]	RX (b ignored)
KX	jumpc0	d,disp[Ra]	RX (d is constant)
RRK	shiftl	Rd,Ra,k	EXP
RkkRk	extract	Rd,e,f,Rg,h	EXP
RCEXP	getctl	Re,Cf	EXP

An EXP instruction may use the fields op, d, ab, e, f, g, h. The g and h fields can be combined into a single 8-bit field gh All EXP instructions combine the a and b fields into a single 8-bit field called ab. Some EXP instructions combine the g and h fields into a single 8-bit field called gh. The EXP format has the following variants.

- The RREXP format takes two register operands, which are in the e and f fields of the second word. The d field of the first word and the g and h fields of the second word are ignored (the assembler will set these to 0). Any RREXP instruction could be reprsented as RRR, but there are only a few RRR opcodes avaiable, so uncommon instructions that require two registers are represented as RREXP. Example: **execute R5,R6** is RREXP.

- The RCEXP format takes two register operands; the first is a general register and the second is a control register. An example of the operand field is **R3,mask**. The operands are in the e and f fields of the second word. The d field of the first word and the g and h fields of the second word are ignored (the assembler will set these to 0.) The first operand is an element of the register file (for example, R4). The second operand is a control register, which is specified by a 4-bit number. In assembly language, we normally refer to the control registers by name rather than number, to make it easier to remember which is which. For example, **getctl R3,status** has RCEXP format.

- The RRREXP format takes three register operands, which are in the f, g, and h fields of the second word. An example of the operand field is $R1,R2,R3*. The d field of the first word and the e field of the second word are ignored (the assembler will set these to 0). The RRREXP instructions would be a natural fit for the RRR format, but there are not enough RRR opcodes available, so the EXP format is used to expand the number of instructions that can be represented. For example, **push R5,R8,R9** has RRREXP format.

- The RRKEXP format takes two register operands and a 4-bit constant number. An example of the operand field is **R1,R2,13**. The register operands are in the f and g fields of the second word, and constant is in the h field of the second word. The d field of the first word and the e field of the second word are ignored (the assembler sets these to 0). For example, **shiftr R3,R6,7** has RRKEXP format.

- The RRKKEXP format takes two register operands and two 4-bit constant binary number operands. The register operands are in the e and f fields of the second word, while the two constants are in the g and h

fields. The d field of the first word is ignored (the assembler sets it to 0).

- The RRXEXP format takes two register operands as well as a memory address specified with an 8-bit offset and index register. Thus these instructions require three registers to be specified, as well as the offset. Thus every bit of both instruction words is needed to represent an RRXEXP format instruction. In assembly language, the memory address is written as **offset[Rh]** where **offset** is an -bit binary number and Rh is a register. The effective memory address is **offset+Rh**. This is similar to ordinary memory addresses; the only difference is that it uses an 8-bit offset rather than a 16-bit displacement. For example, **save R1,R9,2[R14]** has RRKEXP format.

Expressions, values and relocatables

- An expression is syntax: 23, -5, $b23e, struc+5, arrEnd-arrStart
- A value denotes a word (it is a number) and is the result of evaluating an expression
- A value is marked as either relocatable or fixed
- Expressions may occur in
  - Displacement field of an assembly language statement; the value of the expression is placed in the displacement field of the corresponding machine language instruction.
  - If a displacement value is relocatable, its address is recorded in the list of addressess of words to be relocated
  - Right hand side of an equ statement. The value may be fixed or relocatable. The name (the left hand side) is defined as a new identifier, the definition line is the line containing the equ, the value is the evaluation of the right hand side, which may be either fixed or relocatable. Identifiers used in the expression on the RHS have the line number included in their usage lines.
  - But identifiers that appear in an expression (even if relocatable) are not recorded in the relocation list; only displacements are placed in the relocation list.

1. Instruction set

   The following sections describe the instructions in groups organized by their function. Some of the groups contain instructions with different formats. From the programmer's perspective the function is more important, so these groups are useful in finding the right instruction to use. (From the perspective of designing a digital circuit to impleemnt the architecture, the format is essential.)

7.3.2 Pseudoinstructions

7.3.3 data

The data statement specifies a sequence of constants to be placed in consecutive memory locations starting at the location counter, subject to relocation. Its argument is a list of one or more 4-digit hex constants separated by commas.

A long block of data can be broken up into several data statements. Suppose x1, x2, etc are 4-digit hex constants. Then

```
data  x1,x2,x3,x4,x5,x6
```

is equivalent to

```
data x1,x2,x3
data x4,x5,x6
```

Suppose

- The module's relocation constant is r
- The location counter has been set to c
- The i'th constant (counting from 0) in a data statement is x.

Then the linker will set mem[r+c+i] := x.

One point to watch out for is that an assembly language data statement uses $ to indicate that a number is a hex constant (e.g. $03b7) but the object language data statement requires all numbers to be 4-digit hex constants, and does not require (or allow) a preceding $ character

7.4 Directives

A directive is an assembly language statement that gives further information about how to translate the program to object code and how to link the code with other modules. Directirves specify metadata but they don't generate an instruction or constant data.

Statement	Label	Operands	Purpose
identifier	module		Define name of module
	org	expression	Set location counter
identifier	equ	expression	Define value
identifier	import	identifier,identifier	Import value from module
	export	identifier	Export values

7.4.1 module

A program may be organized as a collection of modules, where each module appears in a separate file. When several modules are present, each one needs a unique name. The *module* statement declares the name of the module, which is specified in the label field. There are no operands. The following statement says that this is the object code for module named *abc*:

```
abc   module
```

A *module* statement is optional. If none is present in a file, the module is anonymous. If a file does contain a *module* statement, it must be the first statement in the file, although it may be preceded by comments and blank lines. It is illegal for a file to contain more than one *module* statement.

An anonymous module can import other modules, but other modules cannot import anything exported from an anonymous module. This means, in effect, that an anonymous module is useful only as a main program.

It is good practice for the main program to have a module statement; in effect, this is the name of the program as well as the name of the module.

An assembly language file should have a name of the form basename.asm.txt. If there is a module statement *modname module*, then basename should be *modname*. For example, the file Heapsort.asm.txt might contain the statement *Heapsort module*. If there is no module statement, basename is arbitrary.

7.4.2 import

The import statement states that the value of an identifier is defined in another module. During the assembly of the module containing the import, the identifier is given a provisional value of 0, but this will be replaced by the actual value by the linker. For example,

```
x   import  Mod1,x
y   import  Mod1,abc
```

says that x is a name that can be used in this module, but it is defined in Mod1; y can be used in this module but it is defined in Mod1 under the name abc.

7.4.3 export

An export statement says that the module is making the value of a symbol available for use in other modules, which may import it. The statement takes two operands: the name being exported and the value, which must be a 4-digit hex constant. It makes no difference whether the name is relocatable, as the linker performs any relocation before writing the exported value into other modules that import it. Examples:

```
export  haltcode,0
export  fcn,002c
```

The export statement states that the value of an identifier should be made available for other modules to import. For example, this module defines a function and exports it so other modules can import and call it:

```
Mod1     module
         export fcn

fcn      add    R1,R1,R1
         jump   0[R12]
```

7.5 Assembly listing

The first section of the assembly listing shows each line of the source program. The line number appears first, followed by the memory address that the instruction on this line will be placed in. The address is given as a 4 digit hexadecimal number, and it is binary (not two's complement). Next comes the machine language code generated by the line of source code. If the line contains a two-word instruction, there will be two 4-digit hexadecimal values; for a one-word instruction there will be one hex number, and if the line doesn't produce any code these fields will be blank. After the code, the original source statement appears.

The second section of the assembly listing is the **Symbol Table**. This shows each identifier (or "symbol") that appears in the program, the address allocated for the symbol, the source code line where it was defined, and the source code lines where it was used.

7.5.1 equ

```
codeWrite  equ  2
codeRead   equ  1
```

The expression in an equ can calculate the size of an object:

```
astart     data 5
           data 9
           data 78
aend
asize      equ  aend-astart
```

7.5.2 reserve

The `reserve` statement has one operand, which is a numeric constant. The assembler increments the location counter by the value of the operand.

This can be used to reserve a block of memory, for example to hold a stack or heap data structure.

```
asize   equ    100
n       reserve asize
```

If the operand of a `reserve` statement is a symbol defined by an `equ`, that definition must precede the `reserve` statement. This is because pass 1 of the assembler needs to calculate the address of each code word, and an `equ` that follows the `reserve` statement would not become known until after the size of the `reserve` is needed.

7.5.3 org

The org statement sets the location counter to a specified address. Subsequent instructions and data will be placed in memory at contiguous locations starting from that address.

```
org   35     ; subsequent code starts from address 0023
```

The assembler initializes the location counter to 0 before it begins translating an assembly language module. Every module begins implicitly with org 0.

8 Object code and linker

Small programs often consist of just one module (or file). The assembler translates the assembly language source code into machine language which is then executed by the processor.

However, there are several reasons for breaking up larger programs into several modules. It's easier to work with several modules of reasonable size rather than one gigantic file. A module may provide generic services that can be incorporated into many programs. Programs can be simplified if they use libraries for common tasks, rather than implementing everything from scratch. It is faster to assemble small files than large ones.

When a program consists of several modules, each one can be assembled separately. However, the resulting machine language is not executable if it refers to procedures or other values defined in another module. An instruction

in module A cannot refer to a word X in module B unless it knows the address of X, and when module A is assembled it knows nothing about module B.

To produce an executable program, its modules need to be combined into a single executable module, with all the addresses resolved. This is called **linking**.

Sigma16 supports linking. The system is designed so that programs that consist of just one standalone module can be executed directly, without linking. This means you can ignore the issues of modules and linking if you just want to write a andalone program.

8.1 Object language

Object code is expressed in a textual language, so the object code is readable by a human (at least, by a human who understands machine language). For example, binary data is specified using four hexadecimal characters rather than a word of binary data.

8.1.1 Object statement syntax

The object language has a simple syntax and only a few types of statement. Each object statement is written on one line. It begins with a keyword indicating the type of statement, followed by one or more spaces, followed by an operand field which must not contain any white space. The operand field is a comma-separated list of tokens; each token is either a hex constant or an identifier.

- In the object language, hex constants are written as four characters, using digits 0-9 a-f. Unlike assembly language, a hex constant is not preceded by $. There is no need for this, as all numbers are written in hex in object code. Assembly language allows both hex and decimal numbers so there needs to be a way to tell them apart.

- Identifiers have the same syntax as in assembly language: a string of letters, digits, and underscore characters, beginning with a letter.

The object language has seven statements: `module`, `org`, `data`, `import`, `export`, and `relocate`. Some of these are related to corresponding statements in assembly language, but their syntax is different and in some cases they may contain different information.

8.1.2 module

8.1.3 org

8.1.4 data

data x0,x1,. . . ,x_{j-1}

Let xs be the list of j words in a data statement, and llc is the linker location counter. For each word x, the linker performs:

```
mem[llc] := x
llc := llc + 1
```

8.1.5 import

General form

```
import  modName,externalName,address,field
```

Examples

```
import  Mod2,abc,03c4,dist
import  Mod3,ybit,03be,g
```

8.1.6 export

8.1.7 relocate

The relocate statement specifies a list of addresses of words that must be relocated. Suppose the value x is specified in a relocate statement, and the linker is relocating the module by offset y. Then the linker will set mem[x+y] = obj[x]+y.

```
relocate hex4,hex4,\ldots{}
```

General form:

The relocate statement specifies a list of addresses, which refer to object code words in the module. The effect is to add the linker location counter (llc) to each object code word.

code[addr] := code[addr] + llc

```
relocate  addr,addr,\ldots{},addr
```

Each location is relcated. The word The addresses

8.2 Module metadata

The assembler and linker create metadata files which enable the emulator to show the assembly language statement corresponding to the instruction currently being executed. The metadata is not part of the machine language, and the emulator doesn't look at it in order to execute the program. It is entirely optional: the emulator can run a program without any metadata, although without it the emulator cannot display the current assembly language source statement. This section explains how the metadata works and the format of the files.

The emulator attempts to show the assembly language source as the program runs, and it highlights the current and next instruction. To do this, the emulator needs to have some information that isn't present in the object code. This extra information is supplied in a separate metadata file produced by the assembler and the linker.

An object file foo.obj.txt may have a corresponding metadata file foo.omd.txt ("object metadata"). An executable file foo.exe.txt may have a corresponding metadata file foo.xmd.txt ("executable metadata"). The format of the metadata is identical for object and executable; the reason for the distinction is that the user might have a program with main program foo.asm.txt, and later give the executable the same name foo. In that case, there will be separate metadata files for the object and the executable.

The metadata contains the source code in two forms: plain text and with html tags for highlighting the fields. In addition, the metadata contains a mapping from address to source code line number.

The metadata file format is parsed in order to populate several data structures that enable the emulator to The metadata contains the lines of text of the assembly listing. These lines contain the address, the object code

at that address, and the assembly language source statement. Each line of the assembly listing appears t The emulator displays most lines of the assembly listing with the same field highlighting

A metadata file contains two sections: the ASmap followed by the source listing text. A metadata file must have the following contents:

$a0,s0,a1,s1,$ $a2,s2, \ldots,$ a_{n-1}, s_{n-1}

When the pc contains address a_i then the source statement s_i should be displayed.

- `fsmap` n
- comma separated list of n numbers, which may be split into lines
- `source` n
- n lines of html giving the assembly listing. Each line appears twice: first a "plain" form, followed by a "decorated" form that contains html span elements for highlighting the fields of the text

Here is an example of a metadata file:

```
fasmap 17
14,14,15,15,16,16,17,17,18,19
19,20,20,21,21,22,24
source 32
<span class='ListingHeader'>Line Addr Code Code Source</span>
<span class='ListingHeader'>Line Addr Code Code Source</span>
   1 0000            ; Main: test linker
   1 0000          <span class='FIELDLABEL'> \ldots{} </span>
```

8.3 Linker

- GUI: selected module is main program and also receives the executable. All other modules are linked, and their object code is placed after that of the selected module. It is an error if any module has not been assembled. The order of the object code depends on the order of the modules in the module list, which is essentially arbitrary, except that the selected module always comes first in the executable.

8.4 Booter

8.4.1 Executable code

An executable module is written in the same language as object modules. The only difference is that an executable module must contain only these types of statement: module, data, org. It is now allowed to contain any of the following statments: import, export, relocate.

If an assembly language program doesn't contain any import or export directives, then its object code won't contain any import, export, or relcate statements. In this case, the object code is already executable and does not require linking: it can be booted directly by the processor.

The booter (invoked by clicking the Boot button in the processor page) reads in the currently selected module and checks to see whether it is a valid executable module. If so, it loads the code into the memory. If not, it indicates that the program cannot be booted.

9 Programming

9.1 Structure of a program

Simple ("static") variabls need to be declared with a data statement, which also gives an initial value.

```
x  data  23
```

This means: allocate a word in memory for x and initialize it to 23. The data statements should come after the trap instruction that terminates the program

The software conventions are:

- R14 - stack pointer
- R13 - return address
- R12

9.2 How to perform commmon tasks

9.2.1 Using extract

A special case is to move a Boolean from one place to another.

- A Boolean is a bit in a register, so it takes two 4-bit fields to specify an arbitrary Boolean
- Would like to make it easy to implement b := c, where b and c are arbitrary Booleans
- This would require two 4-bit fields for each of b and c, for a total of four 4-bit fields
- The Exp format could accommodate this
- But this could also be done using the extract instruction
- Therefore it should either be omitted, or else be a pseudo instruction that generates an extract

The extract instruction is not essential: it can be performed by a sequence of shift and logic instructions. However, an extract instruction is faster than the equivalent sequence of shifts and logic, and it also makes a program more readable by making the intention clear.

Pseudoinstruction
Copy a bit
Invert a bit
Generate a field mask

The field pseudoinstruction loads a word into the destination register Rd; this word consists of 1 bits in the specified field (g,h) and 0 in all other bit positions. This provides a field mask that can be used with logic instructions for a variety of purposes.

- General form: **field Rd,g,h**
- Pseudo-instruction: **injecti Rd,R0,g,h**
- Assembler format: RKK

Semantics

- Rd.i = 1 for g <= i <- h
- Rd.i = 0 for i < g or i > h

Example:

```
field   R3,4,  ; R3 := 0fc0
```

Using a field mask

- invert it to give negative mask
- and R1 with mask to clear bits outside the field
- and R1 with negative mask to clear only the field
- xor R1 with mask to invert bits in the field

9.2.2 Copying one register to another

Sometimes you want to copy a value from one register to another: R3 := R12. There isn't an instruction specifically for this purpose, because there is no need: just use the add instruction:

```
add R3,R12,R0 ; R3 := R12
```

Since R12 + 0 = R12, this copies the value in R12 into R3.

9.3 Compilation

There are two ways to handle variables:

The statement-by-statement style: Each statement is compiled independently. The pattern is: load, arithmetic, store. Straightforward but inefficient.

The register-variable style: Keep variables in registers across a group of statements. Don't need as many loads and stores. More efficient. You have to keep track of whether variables are in memory or a register. Use comments to show register usage. Real compilers use this style. Use this style if you like the shorter code it produces.

We'll translate the following program fragment to assembly language, using each style:

```
x = 50;
y = 2*z;
x = x+1+z;
```

Statement-by-statement style

```
; x = 50;
     lea    R1,\$0032   ; R1 = 50
     store  R1,x[R0]    ; x = 50

; y = 2*z;
     lea    R1,\$0002   ; R1 = 2
     load   R2,z[R0]    ; R2 = z
     mul    R3,R1,R2    ; R3 = 2*z
     store  R3,y[R0]    ; y = 2*z

; x = x+1+z;
     load   R1,x[R0]    ; R1 = x
     lea    R2,1[R0]    ; R2 = 1
     load   R3,z[R0]    ; R3 = z
     add    R4,R1,R2    ; R4 = x+1
     add    R4,R4,R3    ; R4 = x+1+z
     store  R4,x[R0]    ; x = x+1+z
```

Register-variable style

```
; Usage of registers
;   R1 = x
;   R2 = y
;   R3 = z

; x = 50;
     lea    R1,\$0032   ; x = 50
     load   R3,z[R0]   ; R3 = z
     lea    R4,\$0002   ; R4 = 2
; y = 2*z;
     mul    R2,R4,R3   ; y = 2*z
; x = x+1+z;
     lea    R4,\$0001   ; R4 = 1
     add    R1,R1,R4   ; x = x+1
     add    R1,R1,R3   ; x = x+z
     store  R1,x[R0]   ; move x to memory
     store  R2,y[R0]   ; move y to memory
```

Comparison of the styles
Statement by statement.

- Each statement is compiled into a separate block of code.
- Each statement requires loads, computation, then stores.
- A variable may appear in several different registers.
- There may be a lot of redundant loading and storing.
- The object code corresponds straightforwardly to the source code, but it may be unnecessarily long.

Register variable

- The instructions corresponding to the statemnts are mixed together.
- Some statements are executed entirely in the registers.
- A variable is kept in the same register across many statments.
- The use of loads and stores is minimised.

- The object code is concise, but it's harder to see how it corresponds to the source code.
- It's possible to have a mixture of the styles: you don't have to follow one or the other all the time.

9.4 Errors: avoiding, finding, and fixing

9.5 Procedure call and return

A common usage of save and restore is to simplify procedure call and return. When a procedure is called, store the registers onto the execution stack when a procedure is called (using save), and then to load them back from the stack when the procedure is returned (restore). Normally, the format of a stack frame has a fixed location for saving the registers, at a small offset from the beginning of the frame. A register called the *stack pointer* gives the address of the frame, and the offset for saving the registers is normally a small value (such as 3 or similar).

9.5.1 Critical regions

A testset instruction is not semantically equivalent to a load followed by a store. Consider this example:

```
; (1) testset
     testset   R1,mutex[R0]
```

It is not the same as

```
; (2) sequence of instructions
     load      R1,mutex[R0]
     lea       R2,1[R0]
     store     R2,mutex[R0]
```

The essential difference is that (1) executes as an atomic operation, but (2) does not, and this could lead to errors in mutual exclusion, which could lead in turn to fatal errors, crashes, and security violations.

Consider, for example, a situation where two processes are sharing mutex to control access to a critical retion.

an interrupt could occor after the load and before the store. Suppose, for example, that initially mutex = 0 and the sequence is executed. Another process could be performing a similar sequence of instructions on the same mutex variable.

9.5.2 Robust programming

*Use a systematic programming process

- Start with a high level algorithm
- Then translate that to the low level ("if b then goto label") form
- Translate the low level to assembly language, keeping the higher level versions as comments

Use comments both to develop the program and to document it

- Write the comments first, as you develop the program. There should already be some good comments (e.g. the algorithm) before any instructions at all have been written.
- Don't fall into the trap of hacking out instructions and then adding comments later: this loses the benefits that documention offers as you're writing the code.

How to write good comments

- Keep the high level and low level algorithms as comments
- Comment each instruction
- Use the comments to explain what your program is doing, not to explain what an instruction does.
- Assume that the reader already knows the language, but not the details of your program.

9.5.3 Error messages

9.5.4 Runtime debugging

What if an instruction doesn't do what you expected?

- Execute the program to the point where the mysterious instruction is about to be executed, but has not yet executed. (To do this, you can step through the program, or set a breakpoint.)
- Make sure you know what the instruction is supposed to do (check the User Guide).
- Looking at the state of the registers and memory, carefully predict what you expect the instruction to do.
- Execute the one instruction (click Step) and compare the state of the machine with your prediction.
- Make sure the instruction has not been modified in memory. Compare the machine language produced by the assembler with the **current** contents of the word or words in memory where the instruction is located.

9.5.5 Breakpoints

(Note: the breakpoint system is not fully implemented yet; the following describes a temporary breakpoint facility.)

A breakpoint is the address of an instruction; when the machine is about to execute that instruction (i.e. when the pc contains that address) the emulator will halt execution, enabling the programmer to examine the state of registers and memory. To set a breakpoint, click Breakpoint and enter the instruction address you want to stop at in the dialogue box. There are several control buttons. Refresh means "read the contents of the text in the box, which must be a $ followed by a 4 hex digit address". Whenever you change the text, you should click Refresh. The Enable button toggles the breakpoint on and off. The Close button hides the Breakpoint dialogue box. Here's an example. Suppose you want to stop execution of a program at address 00f6:

- Click Breakpoint

- Enter $00f6
- Click Refresh
- Click Enable
- Click Close
- Click Run

The execution will run until the pc becomes equal to 00f6 and will then stop.

Click Refresh, then Enable, then Close. Then click Run, and the emulator will run at full speed until the pc reaches the specified value; then it will stop so you can examine the state of the machine.

10 Installation

You can run most of Sigma16 in a web browser – there's nothing to download, nothing to install. Visit the Sigma16 Home Page in your browser:

[Sigma16 home page](`https://jtod.github.io/home/Sigma16/`) Click on the link to launch the app. It will run in your browser; you don't have to install anything. The Home page also contains links to the source code and further information about the project.

10.1 Command line tools

Sigma16 also contains some advanced features that use the command line in a shell. These features include text commands for assembling and linking programs, and for running the circuit simulator. These features require some software installation. In addition, building Sigma16 from soure requires some additional software that must be installed. All of the software required is free and open source, and all of it runs on Windows, Macintosh, and Gnu/Linux.

10.1.1 node and npm

10.1.2 Configuring the shell

Shell running bash

Add the following to your .bashrc file, but replace *Users* yourlogin/ Documents/ path/ to/ with your own file location. In a bash shell running on cygwin, try *Users* yourlogin.

```
SIGMA16=/Users/yourlogin/Documents/path/to/SigmaProject/Sigma16
export SIGMA16
alias helloworld="node \$\{SIGMA16\}/app/helloworld.js"
```

10.1.3 Testing the installation

```
\$ node --version
v16.5.0
```

10.1.4 Building Sigma16

The Web version of Sigma16 contains several files that need to be built from source. Of course, if you launch the app from the Sigma16 Home Page you don't need to worry about that: you get a fully-built version.

Use emacs to build *.html from source *.org

```
\^{}C \^{}E h h
```

```
\$ npm install -g wabt
\$ wat2wasm emcore.wat --enable-threads
```

10.2 In case of problems

If you encounter a problem with the app, please file a bug report. It is essential in a bug report (for any software, not just Sigma16) to provide as much as possible of the following information.

- State what version of the software you are running. This is visible in the Welcome page, as well as the User Guide and the Options page.

- State what browser and operating system you are using. There are some incompatibilities between Chrome, Firefox, Edge, and Safari, as well as differences between operating systems.
- Describe what the problem was.
- Provide the source code of the assembly language program you are running.
- If possible, provide some photos or screen shots showing the app at the point where the problem arose. Smartphone photos are fine. Try to show the processor display, including the registers.

11 About Sigma16

11.1 Copyright and license

The architecture, software tools, and documentation were designed, implemented, and written by John O'Donnell. Contact email: john.t.odonnell9@gmail.com. Copyright (C) 2019-2025 John T. O'Donnell

Sigma16 is available under the GPL-3 license. The license text is available

- in the Sigma16 folder at Sigma16/LICENSE.txt
- online at https://www.gnu.org/licenses/gpl-3.0.en.html

Sigma16 is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, Version 3 of the License. Sigma16 is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with Sigma16. If not, see www. gnu. org/ licenses.

11.2 Release notes

11.2.1 Version 3.5.0, January 2023

- No change to the Core instruction set

- revised the Standard instructions, some changes to representation
- added family of branch instructions

11.2.2 Version 3.4.0

- Bit indexing is changed to "little end" style. The least significant bit has index 0, and the most significant bit has index 15.
- The architecture is organized into precisely defined subsets: Core and Standard

11.2.3 Version 3.3.2, April 2021

- There is no change in the architecture
- The software is modified to enable it to continue working when a planned change to web browsers occurs in May 2021. The program runs on a hosted web server that enables it to work fully with cross origin isolation.

11.2.4 Version 3.2.3, development from April 2021

11.2.5 Version 3.2.2, March 2021

- A bug in breakpoints is fixed
- In addition, there is a new way to specify breakpoings using trap
- When the emulator stops, the memory display is correct; you no longer need to refresh it

11.2.6 Version 3.2.1, February 2021

Version 3.2 brings several changes that will be visible to users of previous versions of Sigma16:

- *cmplt, cmpeq, cmpgt are removed.* Instead, use the `cmp` instruction, which sets the condition code (`R15`), and then use any of the conditional jump instructions `jumplt`, `jumple`, `jumpeq`, `jumpne`, `jumpge`, `jumpgt`. (Rationale: There are more Booleans in the condition code than just

less-than, equal, and greater-than. The new style accomodates all the conditions in a uniform manner, but the old style does not. Version 3.1 already supported `cmp` and the conditional jumps.) Here's an example:

```
; Old style -- these instructions have been removed
     cmplt   R1,R2,R3      ; R1 := R2 < R3
     jumpt   R1,loop[R0]   ; if R2 < R3 then goto loop
; New style -- use the following instead
     cmp     R2,R3         ; compare R2 with R3
     jumplt  loop[R0]      ; if R2 < R3 then goto loop
```

- *jumpf is renamed to jumpz, and jumpt is renamed to jumpnz.* The new names stand for *jump if zero* and *jump if not zero*. Most old programs will use `jumpf` or `jumpt` only after `cmplt`, `cmpeq`, or `cmpgt`, but following `cmp` you should use one of the conditional jumps listed above. (Rationale: The new names reflect more accurately what the instructions do. The decision about whether to jump depends on whether the entire register contains 0; it isn't a decision based on checking just a single bit.)

The following changes do not require modifying old programs; they just relax the syntax rules so some programs would no longer give an error message.

- *[R0] is optional.* In previous versions of Sigma16, every displacement requires the index register to be stated explicitly, even if it's `R0`: for example, `load R3,xyz[R0]`. Now, the `[R0]` can be omitted, although you can include it if you wish. Thus `load R3,xyz` and `load R3,xyz[R0]` are equivalent. (Rationale: The reason for requiring `[R0]` in the past was to emphasise the regularity of the instruction representation. However, a primary aim of the design of Sigma16 is to provide subsetting of the architecture, which supports a spiral approach to learning computer architecture. Another aim is to provide a good platform for schools or other students who will just learn a little of the system. Removing the requirement for `[R0]` simplifies the language for a beginner. Furthermore, for an experienced expert programmer it's more readable to omit the `[R0]` as this reduces the amount of clutter in the code.)

- *Allow lower case "r" in register names.* In previous versions, elements of the register file required an upper case `R`: for example, `R8`. Now you can write `r8` as well as `R8`, and both names refer to the same register. (Of course the same holds for the rest of the register file.) This is a trivial syntax issue. There's no technical reason to prefer `r8` or `R8`; it's just a matter of personal preference. It may be easier to read `r8` because the lower case `r` is shorter than the digits. It's good style to use either the `Rn` or `rn` form consistently, but the assembly language doesn't enforce that. Labels are case sensitive but registers are not. Labels are case-sensitive, so `xyz` and `XYZ` are distinct names, but registers are not labels, and registers are not case sensitive.

There are some changes to the machine language that don't affect assembly language programs.

- Some of the opcodes have changed, so programs need to be reassembled.
- The word logic instructions `inv`, `and`, `or`, `xor` are now pseudoinstructions that generate the `logicf` instruction. The assembly language syntax is the same as before; only the underlying machine language representation is different

There are some new instructions and features, as well as some instructions and features that have been in the architecture for some time but weren't documented in the User Gude. These won't affect existing programs.

110