

4. Instructions

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Machine Code

- A machine code instruction must specify some operation (such as **ADD** or **JUMP**) but also any necessary auxiliary information (*where* to get the operands, store results, where to jump to etc.).
 - Each instruction is held as one or more words in memory (consecutive addresses).
 - The first word of an instruction is called the **operation word** (OW). Some instructions have only an operation word but others require additional words.
 - In the OW is an **op-code**, a set of bits which tells the CU the operation involved.
 - Once the operation is known, the CU must be told where to get operands & where to place the result. This info is stored in the rest of the OW and/or auxiliary words.
- In principle operands may be located in any memory location or register file register, likewise for results, but to prevent instructions becoming too long, restrictions may be imposed on where a particular instruction can access.:
 - For example, some instructions may be restricted to use CPU registers only.

Addressing Modes

- In any CPU design there are different ways, called **addressing modes**, of specifying where an operand for an instruction is to be found or a result to be stored.
 - A source or destinations is either a memory location or CPU r-file register.
 - *The addressing modes allowed vary from one design to another as do their names.*
 - Some memory addressing modes involve calculation by the CPU before the actual or **effective address** (EA) required is known.
- Some common examples with generic names.
 - **Register direct**: names the CPU/register involved.
 - **Absolute**: give the numeric address of the memory loc involved.
 - **Register indirect**: gives number of register where address of memory loc can be found.
 - **Indexed absolute**: gives an absolute **base address** and the number of a register containing an **index address**; EA is the sum of the two; the register is called an **index register** in this role. In some designs any general register can act as an index; in others only certain ones.
 - **Relative**: specifies a signed numeric offset to add to the current PC address.
 - **Immediate** (the operand is in the instruction)
- Not all modes make sense in all situations. They are used not only in instructions involving calculations but also in those transferring program control (e.g. jumps).

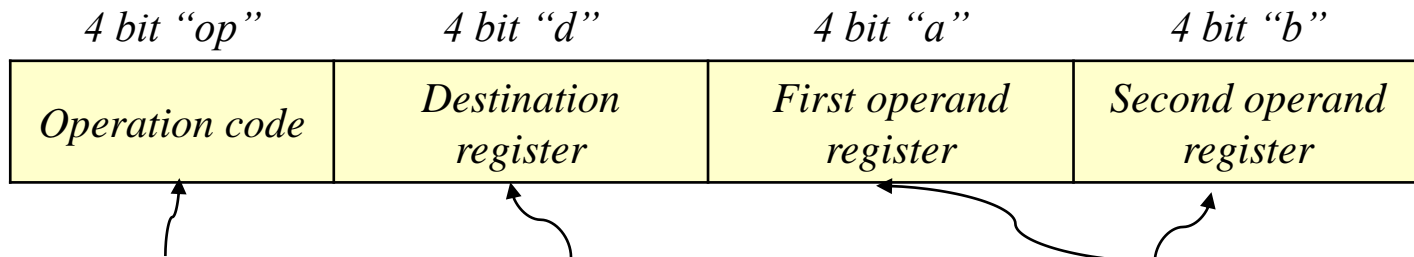
Sigma16

- Sigma16 (S16) programmer's model consists of:
 - The 16 registers R0..R15 (R0 always 0)
 - The PC
 - 65,536 16-bit memory locations with addresses 0000_{16} to $ffff_{16}$.
- There is no status register. Sigma16 has no overflow or other flags.
 - This is a significant ~~shortcoming~~ although not for our purposes.
- A Sigma16 machine code instruction consists of one or two 16 bit values stored in consecutive memory locations.
- The most significant 4-bits of the operation word constitute the **op-field**. There are 16 possible op-codes that can be held in the op-field.
- Sigma 16 has only two addressing modes.
 - **Register addressing** gives a register file register as source or destination
 - **Indexed addressing** gives a register and a 16-bit number; these are added to get the EA (this is just indexed absolute in previous naming)

Sigma16 RRR Instructions

- Most S16 op-codes, those from 0 to D_{16} are in the **RRR family**. All RRR instructions are one word long. All share the same operation word format.
 - First 4 RRR instructions (op-codes 0-3) are arithmetic: **ADD, SUB, MUL, DIV**
 - Next 3 are compare instructions: **CMPLT, CMPEQ, CMPGT**.
 - Next 4 are bitwise Boolean: **INV, AND, OR, XOR**.
 - Next two are shifts: **ASHL, ASHR**.
 - Last (op-code D_{16}) is a special instruction called **TRAP**
- MUL, DIV and compare instructions work only with *2's complement values*.
 - For this reason S16 is good at handling 16-bit two's complement codes; using anything else is much more awkward!
 - Unless otherwise stated all numbers here can be assumed to be 16-bit two's complement.
- RRR instructions only use register addressing (values and results in the reg. file).
- Each RRR instruction specifies 3 regs: first stores result, the next two the operands.
 - E.g. **ADD Rx, Ry, Rz** means add Ry and Rz, result in Rx
 - Note assembly language syntax of ADD statement: this is typical of all RRR instructions

RRR Instruction Format



The "*opcode*" says what kind of instruction this is

Specifies where the result goes

Specifies which registers contain the two operands

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- Each RRR OW consists of four 4-bit fields: *op*, *d*, *a*, *b*.
 - op* contains the op-code
 - d* contains the destination register number (N.B. *R0* cannot be written to as it is always 0)
 - a* and *b* contain the operand register numbers (*b* is ignored in the INV instruction)
- TRAP instructions are used to request service from the operating system. In this course TRAP is only used to terminate a program gracefully and return control to the OS. In this role the *d*, *a* and *b* fields are all set to 0.

Sigma16 Assembly Language I

- A Sigma16 assembly language program consists of a list of **assembly language statements** and **assembler directives**, one to each line.
- An assembly language statement specifies exactly one machine code instruction and consists of 4 fields separated by at least one space and formatted as follows:

Assignment Project Exam Help
`Label Mnemonic Operands ;Comment`

- The **Label** can be any name chosen by the programmer. A line is normally only labelled if it will be referenced by another instruction. A label always begins in the first column.
- The **Mnemonic** is a short name for the instruction (e.g. ADD, SUB etc)
- The **Operand field** lists the other information needed by the instruction (where are the operands, where will the result be)
- The optional **comment field** always begins with a “;” character

Sigma16 Assembly Language II

- For an RRR, the operand field always lists the 3 registers Rd,Ra,Rb (in that order) separated by commas. This form is used even when 3 registers are not needed. E.g.

INV R3,R1,R2

- This reads R1, inverts its bits and stores the result in R3. R2 is ignored.
- In the S16 assembly language numbers preceded by a “\$” are hexadecimal; without this they are decimal. Thus \$100 is 100₁₆ which is 256₁₀.
- An *assembler directive* is not translated to machine code but is an instruction to the assembler program to do something. Commonest example is the DATA directive which allows data to be preset in specific memory. This has the format.

xyz DATA \$3000 ;Comment

- which gives the label xyz to the next available memory location and initialises it to \$3000 (what is this in decimal?)
 - xyz can now be treated as the name of a (16-bit) variable.

Example: A Simple Program

- An assembly language program is just a list of assembler statements.

- Suppose that we have some integer (two's complement) variables in registers: x is in R1, y is in R2, z is in R3.

- We want to compute $x + y \times z$, and leave the result in R4.

- Solution:

MUL

R5, R2, R3

; R5 = $y \times z$

ADD

R4, R1, R5

; R4 = $x + (y \times z)$

- The machine code (in hex) for this program would be as follows:

2523 (Op-code for MUL is 2)

0415 (Op-code for ADD is 0)

- Exercise: Write this out in binary.

Sigma16: RX Instructions

- An instruction with an op-code of F_{16} is called an **RX instruction**.
- An RX instruction has an auxiliary word in addition to the OW. Like RRR the OW is divided into *op*, *d*, *a*, and *b* fields. However here, *b* is used to hold an extension of the op-code.
- This technique, called **expanding op-code**, provides more instructions than the basic op-code allows.
- There are 6 RX instructions: **LEA**, **LOAD**, **STORE**, **JUMPF**, **JUMPT**, **JAL**.
- Each such instruction takes two parameters:
 - The first parameter is specified using indexed addressing, adding the content of a register, specified in the *a* field and a 16-bit constant called the **displacement**, in the auxiliary word. The register, which can be any from the Rfile, is said to be acting as an **index register**; the EA, $Ra+x$ is an **indexed address**.
 - The second parameter also needs a destination register number, in the *d* field
- The assembler form for such an instruction is as follows:

LOAD Rd, x[Ra]

 - This computes an address by adding the content of Ra to x . It then reads from the memory location with that address and copies the content into Rd .
 - Note the assembler syntax which is typical of all RX instructions

Sigma16: X Instructions

- An X format instruction has an op-field value of E_{16} and the OW format is almost identical to RX. However:
 - There is only one X format instruction, JUMP, with $b=3$
 - The destination register is not needed and the d field can be set to 0.
 - The assembly format is:
<https://powcoder.com>
 - After a JUMP the CPU transfers control to the address $Ra+x$ to fetch the next instruction.

LOAD, STORE and LEA

- RRR instructions must use CPU Rfile registers for operands and results. Fast but few.

- To get operands from memory LOAD instruction is used.

`LOAD R1, $3FFF[R0]`

- loads content of memory location with address \$3FFF into R1 recall that R0 is always set to 0

- To store results in memory STORE instruction is used.

`STORE R1, $3FFF[R0]`

- stores content of R1 into memory location with address \$3FFF

- To load a register with a constant, LEA (Load Effective Address), is used. This works out the effective address in its second operand and loads that address (not its content) into the register named as its first. Thus:

`LEA R1, $3FFF[R0]`

- loads the value $3FFF_{16}$ into R1.

Some Examples

- Example: Add 30 and 31. Store the result in loc 100.
 - Solution: Lots of options but here we use registers R1 and R2 to do the adding.
 - We could use a third register for the result or just overwrite R1 or R2. Here we use R3.
 - Note use of LEA instructions to create constants.

```
LEA R1,30[R0]           ;Load R1 with the constant 30
LEA R2,31[R0]           ;Load R2 with the constant 31
ADD R3,R1,R2             ;Add R1 and R2, result in R3
STORE R3,100[R0]         ;Store R3 in location 100 ($64)
TRAP R0,R0,R0            ;Stop
```

- Example: Add the content of loc 30 and loc 31. Store result in loc 100.
 - Solution: As above but now we are adding variables stored in memory. Values are variables loaded using LOAD instruction. R0 (=0) allows us to set *addresses* as constant.

```
LOAD R1,30[R0]
LOAD R2,31[R0]
ADD R3,R1,R2
STORE R3,100[R0]
TRAP R0,R0,R0
```

Note that in a HLL we would not worry about the *addresses* of our variables: we let the compiler choose. The same can be done in assembler using the DATA directive!

- Exercise: Add comments to the second program!

Booleans, Compares and Jumps

- The compare instructions, CMPLT, CMPEQ and CMPGT generate a Boolean result. In S16, FALSE is coded as 0 and TRUE as anything else. For example:

CMPLT R1, R2, R3

- will set R1 to 1 if $R2 < R3$, 0 otherwise.

- The compare instructions are used in conjunction with conditional jumps JUMPF, JUMPT. For example

JUMPF R1, x [R2]

- will jump to address $R2 + x$ but only if R1 is 0; otherwise no jump occurs and the instruction after the JUMPF is executed