ANNA Guide

by Dr. Eric Larson
Seattle University

Table of Contents

Acknowledgments

This document is based on the documentation provided for the ANT assembly language developed at Harvard University, created by the ANT development team consisting of Daniel Ellard, Margo Seltzer, and others. Many elements in presenting their assembly language are used in this document. For more information on ANT, see http://ant.eecs.harvard.edu/index.shtml.

The ANNA assembly language borrows ideas from many different assembly languages. In particular:

- The ANT assembly language from Harvard University. In addition, several of the simulator commands were ideas from the ANT tool suite.

- The LC2K assembly language used in EECS 370 at the University of Michigan.

- The simple MIPS-like assembly language suggested by Bo Hatfield (Salem State College), Mike Rieker (Salem State College), and Lan Jin (California State University, Fresno) in their paper *Incorporating Simulation and Implementation into Teaching Computer Organization and Architecture*. Their paper appeared at the 35th ASEE/IEEE Frontiers in Education Conference in October 2005.

The name ANNA comes from my daughter Anna, who was 6 months at the time when this document was created.

1. ANNA Architecture

This section describes the architecture of the 16-bit ANNA (A New Noncomplex Architecture) processor. ANNA is a very small and simple processor. It contains 8 user-visible registers and an instruction set containing 16 instructions.

1.1 Memory Organization

- Memory is word-addressable where a word in memory is 16 bits or 2 bytes.
- The memory of the ANNA processor consists of 2^{16} or 64 K words.
- Memory is shared by instructions and data. No error occurs if instruction memory is overwritten by the program (your programs should avoid doing this).
- ANNA is a load/store architecture; the only instructions that can access memory are the load and store instructions. All other operations access only registers.

1.2 Register Set

- The ANNA processor has 8 registers that can be accessed directly by the programmer. In assembly language, they are named `r0` through `r7`. In machine language, they are the 3-bit numbers 0 through 7.
- Registers `r1` through `r7` are general purpose registers. These registers can be used as both the source and destination registers in any of the instructions that use source and destination registers; they are read/write registers.
- The register `r0` always contains the constant zero, If an instruction attempts to write a value to `r0` the instruction executes in the normal manner, but no changes are made to the register.
- The program counter (or PC) is a special 8-bit register that contains the offset (or index) into memory of the next instruction to execute. Each instruction is 2 bytes long. Note that the offset is interpreted as an unsigned number and therefore ranges from 0 to 2^{16} - 1. The PC is not directly accessible to the program.

1.3 Execution of Programs

Programs are executed in the following manner:

1.3.1 Initialization

1. Each location in memory is filled with zero.
2. All of the registers are set to zero.
3. The program counter (PC) is set to zero.
4. The program is loaded into memory from a file. See section 6 for information about the program file format.
5. The fetch and execute loop (described in Section 4.2) is executed until the program halts via the halt instruction.

1.3.2 The Fetch and Execute Loop

1. Fetch the instruction at the offset in memory indicated by the PC.
2. Set PC ← PC + 1.
3. Execute the instruction.
    (a) Get the value of the source registers (if any).
    (b) Perform the specified operation.
    (c) Place the result, if any, into the destination register.
    (d) Update the PC if necessary (only for branching or jumping instructions).

1.4 Instruction Formats

Instructions adhere to one of the following three instruction formats:

R-type (`add`, `sub`, `and`, `or`, `not`, `jalr`, `in`, `out`)

15 12	11 9	8 6	5 3	2 0
Opcode	*Rd*	Rs_1	Rs_2	Unused

I6-type (`addi`, `shf`, `lw`, `sw`)

15 12	11 9	8 6	5 0
Opcode	*Rd*	Rs_1	*Imm4*

I8-type (`lli`, `lui`, `bez`, `bgz`)

15 12	11 9	8	7 0
Opcode	*Rd*	Unused	*Imm8*

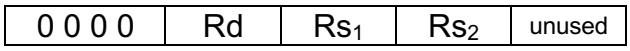
Some notes about the instruction formats:

- The *Opcode* refers to the instruction type. is always in bits 15-12.
- The fields *Rd*, Rs_1, Rs_2 refer to any general purpose registers. The three bits refer to the register number. For instance 0x5 will represent register `r5`.
- The immediate fields represent an unsigned value. The immediate field for `lui` is specified using a signed value but the sign is irrelevant as the eight bits are copied directly into the upper eight bits of the destination register.
- Some instructions do not need all of the fields specified in the format. The value of the unused fields are ignored and can be any bit pattern.
- The same register can serve as both a source and destination in one command. For instance, you can double the contents of a register by adding that register to itself and putting the result back in that register, all in one command.

2. ANNA Instruction Set

In the descriptions below, R(3) refers to the content of register `r3` and M(0x45) refers to the content of memory location 0x45. The descriptions do not account for the fact that writes to register `r0` are ignored – this is implicit in all instructions that store a value into a general-purpose register.

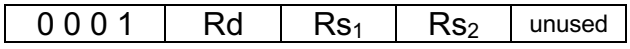
`add` Add



Two's complement addition. Overflow is not detected.

R(*Rd*) ← R(Rs_1) + R(Rs_2)

`sub` Subtract



Two's complement subtraction. Overflow is not detected.

R(*Rd*) ← R(Rs_1) - R(Rs_2)

`and` Bitwise and

Bitwise and operation.

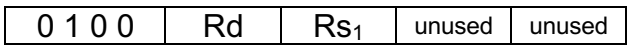
R(*Rd*) ← R(Rs_1) & R(Rs_2)

`or` Bitwise or

Bitwise or operation.

R(*Rd*) ← R(Rs_1) | R(Rs_2)

`not` Bitwise not



Bitwise not operation.

R(*Rd*) ← ~R(Rs_1)

shf	Bit shift	0 1 0 1	Rd	Rs_1	Imm6

Bit shift. It is either left if *Imm6* is positive or right if the contents are negative. The right shift is a logical shift with zero extension.

if (*Imm6* > 0)
    R(*Rd*) ← R(Rs_1) << *Imm6*
else
    R(*Rd*) ← R(Rs_1) >> *Imm6*

lli	Load lower immediate	0 1 1 0	Rd		Imm8

The lower bits (7-0) of *Rd* are copied from the immediate. The upper bits (15- 8) of *Rd* are set to bit 7 of the immediate to produce a sign-extended result.

R(*Rd*[15..8]) ← *Imm8*[7]
R(*Rd*[7..0]) ← *Imm8*

lui	Load upper immediate	0 1 1 1	Rd		Imm8

The upper bits (15- 8) of *Rd* are copied from the immediate. The lower bits (7-0) of *Rd* are unchanged. The sign of the immediate does not matter – the eight bits are copied directly.

R(*Rd*[15..8]) ← *Imm8*

lw	Load word from memory	1 0 0 0	Rd	Rs_1	Imm6

Loads word from memory using the effective address computed by adding Rs_1 with the signed immediate.

R(*Rd*) ← M[R(*Rs1*) + *Imm6*]

sw	Store word to memory	1 0 0 1	Rd	Rs_1	Imm6

Stores word into memory using the effective address computed by adding Rs_1 with the signed immediate.

M[R(Rs_1) + *Imm6*] ← R(*Rd*)

bez	Branch if equal to zero	1 0 1 0	Rd		Imm8

Conditional branch – compares *Rd* to zero. If *Rd* is zero, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. The immediate is treated as a signed value.

if (R(*Rd*) == 0) PC ← PC + 1 + *Imm8*

bgz	Branch if greater than zero	1 0 1 1	Rd		Imm8

Conditional branch – compares *Rd* to zero. If *Rd* is strictly greater zero, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. The immediate is treated as a signed value.

if (R(*Rd*) > 0) PC ← PC + 1 + *Imm8*

addi	Add immediate	1 1 0 0	Rd	Rs_1	Imm6

Two's complement addition with a signed immediate. Overflow is not detected.

R(*Rd*) ← R(Rs_1) + *Imm6*

jalr	Jump and link register	1 1 0 1	Rd	Rs_1	unused	unused

Jumps to the address stored in register *Rd* and stores PC + 1 in register Rs_1. It is used for subroutine calls. It can also be used for normal jumps by using register `r0` as Rs_1.

R(Rs_1) ← PC + 1
PC ← R(*Rd*)

in	Get word from input	1 1 1 0	Rd	unused	unused	unused

Get a word from user input.

R(*Rd*) ← input

out	Send word to output	1 1 1 1	Rd	unused	unused	unused

Send a word to output. If *Rd* is `r0`, then the processor is halted.

output ← R(*Rd*)

3. ANNA Assembler Reference

3.1 Running the Assembler

The ANNA assembler has two panes: an editor pane and an output window. Use the editor pane to type in programs. The output window displays informational messages. The editor has support for standard Windows operations such as cut, copy, paste, print, and undo.

To assemble the program, select the Assemble option from the Assemble menu. You will be asked to save the file if it has been modified since it was last saved. All assembly code files must be saved with a `.ac` suffix.

On success, the assembler with output a file with machine code. The machine code file will have the same name in the same directory as the assembly code file, except that it will have `.mc` suffix. The file consists of a hexadecimal representation of the machine code for your program with the first line referring to the first instruction of the assembly file.

If the assembly file contains a syntax error, the first such error will be displayed in the output window. The line where the error occurs will be selected in the editor pane.

3.2 Assembly Language Format Rules

When writing assembly language programs, each line of the file must be one of…

- blank line (only white space)
- comment line (comment optionally preceded by white space)
- instruction line

An instruction line must contain exactly one instruction. Instructions cannot span multiple lines nor can multiple instructions appear on the same line. An instruction is specified by the opcode and the fields required by the instruction. The order of the fields is the same as the order of the fields in machine code (from left to right). For example, the order of the fields for subtract are `sub` *Rd* Rs_1 Rs_2. The opcode and fields are separated by white space. Only fields that are necessary for the instruction can be specified. For instance, the `in` instruction only requires *Rd* to be specified so it is incorrect to specify any other fields.

Additional rules:

- Opcodes are specified in completely lower case letters.
- A register can be any value from: `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`.
- Register `r0` is always zero. Writes to register `r0` are ignored.

3.2.1 Comments

Comments are specified by using '#'. Anything after the '#' sign on that line is treated as a comment. Comments can either be placed on the same line after an instruction or as a standalone line.

3.2.2 Assembler directives

In addition to instructions, an assembly-language program may contain directions for the assembler. There are two directives in ANNA assembly:

`.halt`: The assembler will emit an `out` instruction with *Rd* equal to `r0` (`0xF000`) that halts the processor. It has no fields.

`.fill`: Tells the assembler to put a number into the place where an instruction would normally be stored. It has one field: the 16-bit signed immediate to be emitted. For example, the directive "`.fill 32`" puts the value 32 where the instruction would normally be stored.

3.2.3 Labels

Each instruction may be preceded by an optional label. The label can consist of letters, numbers, and underscore characters and is immediately followed by a colon (the colon is not part of the label name). No whitespace is permitted between the first character of a label and the colon. A label must appear on the same line as an instruction. Only one label can appear before an instruction.

3.2.4 Immediates

Many instructions and the `.fill` directive contains an immediate field. An immediate can be specified using decimal values, hexadecimal values, or labels.

- Decimal values are signed. The value of the immediate must not exceeds the range of the immediate (see chart below).
- Hexadecimal values must begin with "`0x`" and may only contain as many digits (or fewer) as permitted by the size of the immediate. For instance, if an immediate is 8 bits, only two hex digits are permitted. Immediates with fewer than the number of digits will be padded with zeros on the left.
- Labels used as immediates must be preceded by an '`&`' sign. The address of the label instruction is used to compute the immediate. The precise usage varies by instruction:
  `.fill` directive: The entire 16-bit address is used as the 16-bit value.
  `lui` and `lli`: A 16-bit immediate can be specified. The appropriate 8 bits of the address (upper 8 bits for `lui`, lower 8 bits for `lli`) are used as an immediate.
  `bez` and `bgz`: The appropriate indirect address is computed by determining the difference between PC+1 and the address represented by the label. If the difference is larger than the range of an 8-bit immediate, the assembler will report an error.
  `addi`, `shf`, `lw`, `sw`: Labels are not permitted for 6-bit immediates.

This table summarizes the legal values possible for immediate values:

Opcode	*Decimal Min*	*Decimal Max*	*Hex Min*	*Hex Max*	*Label Usage*
`.fill`	-32,768	32,767	`0x8000`	`0x7fff`	address
`lui, lli`	-32,768	32,767	`0x80`	`0x7f`	address
`bez, bgz`	-128	127	`0x80`	`0x7f`	PC-relative
`addi, shf, lw, sw`	-32	31	`0x00`	`0x3f`	not allowed

3.3 Error Checking

Here is a list of the more common errors you may encounter:

- improperly formed command line
- use of undefined labels
- duplicate labels
- immediates that exceed the allowed range
- invalid opcode
- invalid register
- invalid immediate value
- illegally formed instructions (not enough or too many fields)

4. ANNA Simulator Reference

4.1 Running the Simulator

The first step to using the simulator is to load a program by clicking the Load button. This will prompt the user for the location of a .mc (machine code) file that was generated from the ANNA assembler. The program will be displayed in the code pane.

4.2 Running a Program

The other four control buttons control the execution of a program.

RESET: Resets the program to its original state. Registers all contain 0 (including PC), and memory only contains the data loaded from the machine code file.

STOP: Forces a breakpoint – used to stop execution of a program in an infinite loop.

CONTINUE: Runs the program until the program halts, hits a breakpoint, or is stopped by the user pressing the stop button.

NEXT: Executes one instruction.

Additional notes:

- When asked to enter a value using the `in` instruction, you must enter a 16 bit signed decimal value (-32,768 to 32,767) or hexadecimal value (`0x8000` to `0x7fff`).
- Output values from the `out` instruction will appear in the output window.
- When the program executes a halt instruction, it enters halt mode. In halt mode, you cannot execute an instruction or continue execution until you reset the program.
- The shortcut keys 'Alt-R', 'Alt-C', and 'Alt-N' can be used to reset, continue, and execute next instruction respectively. Menu items also exist for these commands under the Simulate menu.

4.3 Displaying Data

The register pane displays the current value of all the registers including the PC.

The memory pane can display the contents of up to five memory addresses. To view the contents of a memory address, simply type the address in one of the five address boxes. The current value will then be displayed in the corresponding value box. The address must be specified in decimal (unsigned value from 0 to 65,535) or hexadecimal (`0x0` to `0xffff`). The value will be updated appropriately while the program runs.

4.4 Setting Breakpoints

Breakpoints provide a way to stop execution at any point in the program. The typical use is to set a breakpoint at the start of an interesting part of the program, and then to select CONTINUE to run the program up to that point. The program will execute until the instruction at the address of the breakpoint is about to be executed, and then stop.

Up to four breakpoints can be set. To set a breakpoint, simply type an address in one of the breakpoint boxes and check the "En" (enable) box next to it. When the PC is equal to any of the enabled breakpoints, the simulator will stop. Breakpoints are also illustrated with an arrow in the code display for ease of reference.

There are three buttons in the breakpoint pane:

Enable all – Enables all breakpoints.
Disable all – Disables all the breakpoints.
Clear all – Clears all the breakpoints.

Additional notes:

- Breakpoints are automatically cleared when a new program is loaded.
- To change the value of a breakpoint, it must be first disabled.
- The breakpoint address must be specified in decimal (unsigned value from 0 to 65,535) or hexadecimal (`0x0` to `0xffff`).

5. Style Guide

5.1 Commenting Convention

Your program should include the following comments:

- A block comment with your name, name of the program, and a brief description of the program.
- For each function (including the "main" body): indicate what the code does and how each register is used.
- For each function, make special mention of what registers (or memory) are used as input parameters (including the return address) and/or output parameters. Also mention which registers are used as callee-save registers (all other registers will be assumed to be caller-save registers).
- Place a brief comment for each logical segment of code. Since assembly language programs are notoriously difficult to read, good comments are absolutely essential!
    - You may find it helpful to add comments that paraphrase the steps performed by the assembly instructions in a higher-level language.
- A comment that indicates the start of each section (code, data, stack). If a section is not needed in the program, the comment can be omitted.
- Place a brief comment for every variable in the data section.

5.2 Register Usage

- You are free to assign registers as you see except that register `r7` must be the stack pointer if you are using a stack.
- Do not use registers to store global variables. Exception: stack pointer – register `r7`.

5.3 Other Style Guidelines

This section lists some additional style guidelines

- Make label names as meaningful as possible. It is expected that some labels for loops and branches may be generic.
- Use labels instead of hard coding addresses. You do not want to change your immediate fields if you add a line.
- Do not assume an address will appear "early" in the program. An `lli` instruction with a label should always be followed with an `lui` instruction with the same label.
- Indent all lines so lines with labels are not staggered with the rest of the code.
- Use `.halt` to halt the program.
- There is no reason to use `.fill` in the code section. There is no reason to use anything but `.fill` in the data section.