ANNA Reference Guide (Version 2.0)

by Eric Larson
Seattle University

Table of Contents

This document refers to version 2.0 of ANNA. Version 2.0 was created in September 2022.

Last revised: October 24, 2022

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.

I would like to acknowledge to former Seattle University students Seung Chang Lee and Moon Ok Kim who helped create the ANNA assembler and simulator tools.

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 20 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	Function code

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

15 12	11 9	8 6	5 0
Opcode	*Rd*	Rs_1	*Imm6*

I8-type (`lli`, `lui`, `beq`, `bne`, `bgt`, `bge`, `blt`, `ble`)

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

Some notes about the instruction formats:

- The *Opcode* refers to the instruction type and is always in bits 15-12.
- The *Function code* is used by the following instructions, all share the same opcode of 0000: `add` (000), `sub` (001), `and` (010), `or` (011), `not` (100)
- The operands *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`.
- Some instructions do not need all of the operands specified in the format. The value of the unused operands 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

0 0 0 0	Rd	Rs_1	Rs_2	0 0 0

Two's complement addition. Overflow is not detected.

R(*Rd*) ← R(Rs_1) + R(Rs_2)

`sub` Subtract

0 0 0 0	Rd	Rs_1	Rs_2	0 0 1

Two's complement subtraction. Overflow is not detected.

R(*Rd*) ← R(Rs_1) - R(Rs_2)

`and` Bitwise and

0 0 0 0	Rd	Rs_1	Rs_2	0 1 0

Bitwise and operation.

R(*Rd*) ← R(Rs_1) & R(Rs_2)

`or` Bitwise or

0 0 0 0	Rd	Rs_1	Rs_2	0 1 1

Bitwise or operation.

R(*Rd*) ← R(Rs_1) | R(Rs_2)

`not` Bitwise not

0 0 0 0	Rd	Rs_1	unused	1 0 0

Bitwise not operation.

R(*Rd*) ← ~R(Rs_1)

`jalr` Jump and link register

0 0 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

0 0 1 0	Rd	unused	unused	unused

Get a word from user input.

R(*Rd*) ← input

`out` Send word to output

0 0 1 1	Rd	unused	unused	unused

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

output ← R(*Rd*)

`addi` Add immediate

0 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*

`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*

lw	Load word from memory	0 1 1 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	0 1 1 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*)

lli	Load lower immediate	1 0 0 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	1 0 0 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*

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

Conditional branch – compares *Rd* to zero. If R(*Rd*) = 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.

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

bne	Branch if not equal to zero	1 0 1 1	Rd		Imm8

Conditional branch – compares *Rd* to zero. If R(*Rd*) ≠ 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.

if (R(*Rd*) ≠ 0) PC ← PC + 1 + *Imm8*

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

Conditional branch – compares *Rd* to zero. If R(*Rd*) > 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.

if (R(*Rd*) > 0) PC ← PC + 1 + *Imm8*

bge	Branch if greater than or equal to zero	1 1 0 1	Rd		Imm8

Conditional branch – compares *Rd* to zero. If R(*Rd*) ≥ 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.

if (R(*Rd*) ≥ 0) PC ← PC + 1 + *Imm8*

blt	Branch if less than to zero	1 1 1 0	Rd		Imm8

Conditional branch – compares *Rd* to zero. If R(*Rd*) < 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.

if (R(*Rd*) < 0) PC ← PC + 1 + *Imm8*

ble	Branch if less than or equal to zero	1 1 1 1	Rd		Imm8

Conditional branch – compares *Rd* to zero. If R(*Rd*) ≤ 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.

if (R(*Rd*) ≤ 0) PC ← PC + 1 + *Imm8*

3. ANNA Assembly Convention

3.1 ANNA Calling Convention

This section is only relevant for programs that employ function calls.

- The start of the stack is at address `0x8000`. The program is responsible for initializing the stack and frame pointers at the beginning of the program.
- Register usage:
  - `r4`: return value after a function call.
  - `r5`: return address at the beginning of the function call.
  - `r6`: frame pointer throughout the program
  - `r7`: stack pointer throughout the program

- All parameters must be stored on the stack (registers are not used).
- The return value is stored in `r4` (stack is not used).
- Caller must save values in `r1-r5` they want retained after a function (caller save registers).
  - The return address in `r5` is treated like any other caller save register.

- All activation records have the same ordering.
  - First entry (offset 0) is for the previous frame pointer
  - The next *n* entries (offset 1…*n*) are for the function parameters (in the same order as they appear).
  - Remaining entries are used for local variables and temporary values (order left up to programmer).

- Activation record for "main" only has local variables and temporary values.
  - No previous frame
  - No parameters

3.2 ANNA Heap Management

This section is only relevant for programs that employ dynamic memory allocation.

- Dynamic memory in ANNA is simplified – only allocations (no deallocations)
- Heap management table is implemented using a single pointer called `heapPtr`: it points to the next free word in memory.
- Heap is placed at the very end of the program:

```
# heap section
heapPtr:  .fill &heap
heap:     .fill 0
```

4. ANNA Assembler Reference

4.1 Assembly Language Files

Assembly language files are text files and by convention have the suffix `.ac`. Any editor (such as Notepad) can be used to edit assembly language files.

4.2 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 operands required by the instruction. The order of the operands is the same as the order of the operands in machine code (from left to right). For example, the order of the operands for subtract are `sub` *Rd* Rs_1 Rs_2. The opcode and operands are separated by white space. Only operands 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 operands.

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.

4.2.1 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.

4.2.2 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 operands.

`.fill`: Tells the assembler to put a number into the place where an instruction would normally be stored. It has one operand: 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.

4.2.3 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.

4.2.4 Immediates

Many instructions and the `.fill` directive contains an immediate. 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`: The entire 16-bit address is used as the 16-bit value.
    - `lui` and `lli`: The appropriate 8 bits of the address (upper 8 bits for `lui`, lower 8 bits for `lli`) are used as an immediate.
    - *branches*: 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
branches	-128	127	`0x80`	`0x7f`	PC-relative
`addi, shf, lw, sw`	-32	31	`0x00`	`0x3f`	not allowed

4.3 Error Checking

The assembler will check for the errors. Here is a list of the more common errors you may encounter:

- use of undefined labels
- duplicate labels
- invalid instruction name
- invalid register
- invalid immediate value
- immediates that exceed the allowed range
- illegally formed instructions
- incorrect number of operands
- empty file (no instructions)

If any occur, the assemble process is on considered to have failed and the program is unable to be simulated.

The assembler will also emit a warning if the program does not contain a `.halt` directive.

5. ANNA Simulator Reference

5.1 Running the Assembler

To write an assembly file, use any text editor (such as Notepad).

When you are ready to assemble:
1. Select "Choose File" to open a window used to select the file.
2. Click "Assemble File".

The *Assembler Output* window will display an error if there is an error with the assembly file or a message indicating success.

If successful, the program will populate the *Code* window and be loaded into memory. The simulator is able to start.

5.2 Running the Simulator

To run the simulator, there are four control buttons:

- *Run / Continue*: Runs the program until a breakpoint or the program halts.
- *Step (one instruction)*: Executes a single instruction.
- *Reset Simulator*: Resets the program back to the initial state.
- *Clear All Breakpoints*: Removes all breakpoints.

The simulator can be in one of six states:

- NOT LOADED: A program has not been successfully assembled and loaded into memory. The simulator is inactive until a program has been loaded.
- READY: A program has been loaded and is in the initial state (PC is at 0, all registers have 0, etc.). The simulator is active.
- RUNNING: A program is in the middle of execution and has stopped due to a breakpoint or by stepping one instruction at a time. The simulator is active.
- HALTED: The program encountered a halt, terminating the program. The simulator is inactive and must be reset to rerun the program.
- WAITING FOR INPUT: The program is currently executing an `in` instruction. The user must enter a valid input value in order to continue. The user can also reset the simulator.

- ERROR: The simulator encountered an error. This is likely due to a bug in the simulator – contact your instructor. The simulator is inactive and must be reset to rerun the program.

Additional notes:

- When entering an input 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.
- The simulator will stop every 2,000 instructions even if no breakpoints are set. This is used to stop an infinite loop or a program that jumped out of the code section.

5.3 Displaying Data

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

The Memory pane can display the contents of up to four memory addresses. To view the contents of a memory address, simply type the address in one of the four 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 while the program runs.

5.4 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 *Run/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.

To set a breakpoint, simply click the BP check box by the instruction such that the box is checked. When the PC is equal to any of the enabled breakpoints, the simulator will stop.

To clear a breakpoint, click the BP check box such that box is unchecked. All breakpoints can be cleared by pressing the *Clear All Breakpoints* button. Breakpoints are automatically cleared when a new program is loaded.

5.5 Known Issues

A list of known issues is displayed in the More Information page of the simulator.

If you do encounter an issue that is not listed there, contact the instructor. If the simulator goes into an unusable state, the likely workaround is to refresh the browser page which will reset the simulator.

The simulator may not work on browsers or networks that disable JavaScript or file operations. The simulator does not provide any feedback or error messages when this occurs.

6. Style Guide

6.1 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. It's also helpful to list the activation record order and how registers are used.
- 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 a new section.
- Place a brief comment for every variable in the data section.

6.2 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 values 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.
- 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 or heap sections.