ANNA+ Reference Guide
(ANNA Version 3.0)

by Stephen Riley
Seattle University

Table of Contents

This document refers to version 3.0 of ANNA, called ANNA+. Version 3.0 was created in October 2024.

ANNA+ Acknowledgments

ANNA+ is an enhancement of the ANNA 2.0 specification and simulator (see *ANNA 2.0 Acknowledgment*, below.) The additional instructions, directives, and pseudo-ops were defined by Stephen Riley, Adjunct Instructor at Seattle University.

The intent of ANNA+ was to develop command-line utilities for ANNA to expose Computer Science Fundamentals Certificate students to low-level development tools, including:

- command line assembler
- minimal debugger in the style of gdb
- VT100 terminal debugger
- TinyC compiler

The utilities take the form of a dotnet tool that can be found at github.com/stephen-riley/AnnaSim.

ANNA 2.0 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 Eric Larson's daughter Anna, who was 6 months at the time when the original document was created.

Eric Larson 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 22 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 (though 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, PC, is a special 8-bit register that contains the offset (or index) into memory of the next instruction to execute. Each instruction is one word (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 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 terminates via the `halt` pseudo-op.

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`, `outn`, `outs`)

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*	*Imm4*

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 fields *Rd*, *Rs_1*, *Rs_2* refer to any general-purpose registers. The three bits refer to the register number. For instance, 0x5 (0b101) 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 the fields specified in the format. The values 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

2.1 Core instructions

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(Rs1) + R(Rs2)

`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(Rs1) - R(Rs2)

`and`	Bitwise and	0 0 0 0	Rd	Rs_1	Rs_2	0 1 0

Bitwise and operation.

R(Rd) ← R(Rs1) & R(Rs2)

`or`	Bitwise or	0 0 0 0	Rd	Rs_1	Rs_2	0 1 1

Bitwise or operation.

R(Rd) ← R(Rs1) | R(Rs2)

`not`	Bitwise not	0 0 0 0	Rd	Rs_1	unused	1 0 0

Bitwise not operation.

R(Rd) ← ~R(Rs1)

`mul`	Multiply	0 0 0 0	Rd	Rs_1	Rs_2	1 0 1

Two's complement multiplication. Overflow is not detected.

R(Rd) ← R(Rs1) × R(Rs2)

`div`	Divide	0 0 0 0	Rd	Rs_1	Rs_2	1 1 0

Two's complement integer division. Overflow is not detected.

R(Rd) ← R(Rs1) ÷ R(Rs2)

`mod`	Modulo	0 0 0 0	Rd	Rs_1	Rs_2	1 1 1

Two's complement modulo. Overflow is not detected.

R(Rd) ← R(Rs1) % R(Rs2)

`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 Rs1. It is used for subroutine calls. It can also be used for normal jumps by using register r0 as Rs1.

R(Rs1) ← 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	0 0 0

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

output ← R(Rd)

`outn`	Print integer as string	0 0 1 1	Rd	unused	unused	0 0 1

Prints an integer to the debugger output or STDOUT in the runner.

`outs`	Print string	0 0 1 1	Rd	unused	unused	0 1 0

Prints a NUL-terminated string at address M(Rd) to the debugger output or STDOUT in the runner.

`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(Rs1) + 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(Rs1) << Imm6 else
R(Rd) ← R(Rs1) >> 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 Rs1 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 Rs1 with the signed immediate.

M[R(Rs1) + Imm6] ← R(Rd)

`lli`	Load lower immediate	1 0 0 0	Rd	x	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	x	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	x	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 0	Rd	x	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	x	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	x	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	x	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	x	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

2.2 Pseudo-ops

A *pseudo-op* is an instruction that is not obviously supported by the ANNA CPU. Instead, a pseudo-op is often an alias for one or more instructions.

`br`	Branch		unused	x	Imm8

Inserts `beq r0 Imm8` so that branch is always taken.

PC ← PC + 1 + Imm8

halt	Halt processing

Rd	Imm16

Inserts `lli` and `lui` instructions into the assembly stream such that Imm16 is loaded into R(*Rd*). May be used with labels to load an address into R(*Rd*).

R(*Rd*) ← Imm16

jmp	Jump

Rd	unused	unused	unused

Inserts `jalr Rd r0` to jump to the address stored in register Rd.

PC ← R(Rd)

lwi	Load word immediate

Rd	Imm16

Inserts `lli` and `lui` instructions into the assembly stream such that Imm16 is loaded into R(*Rd*). May be used with labels to load an address into R(*Rd*).

R(*Rd*) ← Imm16

mov	Copies the contents of one register to another.

Rd	Rs_1	unused	unused

Bitwise not operation.

R(*Rd*) ← R(*Rs1*)

push	Pushes a value onto the stack.

Rsp	Rs_1	unused	unused

Assembles `sw` and `addi` instructions to push R(Rs_1) to M(*Rsp*) and decrement R(*Rsp*).

M(*Rd*) ← R(Rs_1)
R(*Rsp*) ← R(*Rsp*) - 1

pop	Pops a value from the stack.	Rsp	Rd	unused	unused

Assembles `addi` and `lw` instructions to pop M(*Rsp*) to R(*Rd*) increment R(*Rsp*).

R(*Rsp*) ← R(*Rsp*) + 1
M(*Rd*) ← R(Rs_1)

3. ANNA Assembly Convention

3.1 ANNA Calling Convention

- 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.
  - Function parameters are pushed onto the stack, accessed via FP+*n*.
  - First entry (offset 0) is for the previous frame pointer
  - Next entry (offset -1) is for return address
  - 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

- Alternatively, global variables may be stored in regular memory as labels on `.fill` directives.

3.2 ANNA Heap Management

- 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 `.asm`. 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 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.
- The addi, lw, and sw instructions support a register-offset notation: you don't need to separate Rs_2 and Imm6 with whitespace. Instead, you can use expressions like "`lw`

r1 r6+1". This makes code that uses a frame pointer (FP) to access local variables much more intuitive, as in "sw r1 rFP-2".

4.2.1 Comments

There are two syntaxes for comments:

- Single line comments 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.
- Block comments using `/* … */`: Any lines of text between the `/*` and `*/` delimiters are ignored. The opening `/*` delimiter must be the first non-whitespace on a line; the closing `*/` delimiter must be the last non-whitespace on a 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:

`.cstr`: Tells the assembler to put a NUL-terminated string into memory starting at the current location.

`.def`: Assigns a value to a label, such as "`stack: .def 0x8000`".

`.fill`: Tells the assembler to put numbers into memory starting at the current location. For example, the directive "`.fill 32 0x41`" puts the values 32 and 65 into memory.

`.frame`: Allows the debugger to understand the stack frame (activation record) for a function. See Section 4.4 for more information.

`.halt`: The assembler will emit an `out` instruction with *Rd* equal to `r0` (`0x3000`) that halts the processor. It has no fields. Maintained for backward compatibility with ANNA 2.0; use the `halt` pseudo-op instead.

`.org`: Specifies the address at which all following assembly occurs. Typically this will be `.org 0x0000` in a program. (The assembler defaults to 0x000 as the originating address.)

`.ralias`: Assigns an alias to a register, such as "`.ralias rSP r7`". All aliases must begin with `r`. If a register is exclusively used for a single role, it is recommended that you alias it; eg. for `rSP` for a stack pointer and `rFP` for a frame pointer.

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 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.
- Binary values must begin with "`0b`" and may only contain as many digits (or fewer) as permitted by the size of the immediate.
- 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`, and `lwi`: 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.
    - 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

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)

4.4 Stack frame definitions

When debugging subroutines, it is often helpful to be able to examine the current stack frame (activation record) as defined by a Frame Pointer (FP), which is `r6` by convention.

With the `.frame` directive, you may define a stack frame for a subroutine in such a way that the debugger will recognize when the PC is within the boundary of the subroutine and be able to show the current frame with symbols.

Each stack frame has a *definition* that lays out the structure of the stack frame relative to the Frame Pointer, and a *region* that defines where it is active.

There are three usages for the `.frame` directive:

- `.frame "`*frame definition*`"` defines the structure of the stack frame—see section 4.4.1, below, for the syntax of a frame definition.
- `.frame "on"` defines the beginning of a stack frame's region.
- `.frame "off"` defines the end of a stack frame's region.

4.4.1 Frame definition syntax

The definition of a stack frame is a string that contains the name of the subroutine followed by a list of frame entries and their offsets.

The string is defined as such:

subroutine name | *offset* ~ *entry name* | *offset* ~ *entry name* | …

For example, let's assume we have a subroutine with the following structure:

```
int fib(int n) {
    int res;
    // body
    return res;
}
```

`n`	FP+1
`prev FP`	FP+0
`ret addr`	FP-1
`res`	FP-2

The frame definition would be:

```
.frame "fib|1~n|0~prev FP|-1~ret addr|-2~res"
```

The first element of the string must be the subroutine name; thereafter, the fields do not need to be in numeric sorted order.

4.4.2 Full stack frame example

Assume the following C-like function:

```
int mul2(int n) {
    return n*2;
}
```

With `.frame` directives, this could be compiled as:

```
# function `int mul2(int n)`
#  FP+1  n
#  FP+0  previous FP
#  FP-1  return addr

            .frame  "mul2|1~n|0~prev FP|-1~ret addr"
mul2:       push    r7 r6       # cache FP
            addi    r6 r7 +1    # set up new FP
            push    r7 r5       # push return address
            addi    r7 r7 -1    # create space for stack frame

            .frame  "on"
mul2_body:  lw      r1 r6 +1    # load n from FP+1
            add     r4 r1 r1    # r4 = n*2

            .frame  "off"
mul2_exit:  lw     r5 r6 -1     # load return addr from FP-1
            lw     r6 r6 0      # restore previous FP
            addi   r7 r7 4      # collapse stack frame
            jmp    r5           # return from function
```

5. ANNA Simulator Reference

The ANNA+ simulator may be found at github.com/stephen-riley/AnnaSim. See the README there for build and installation instructions.

When fully installed, run `annasim` to see brief command line instructions.

5.1 Running the Assembler

To write an assembly file, use any text editor such as Visual Studio Code or Notepad.

Then simply run:

```
annasim your_filename.asm
```

The output will be an assembled memory file on the terminal screen (`STDOUT`), which by itself is not that interesting. To write the memory file to your hard drive, use the `-m` switch:

```
annasim your_filename.asm -m your_memfile.mem
```

If you'd like to see what the assembled bits look like in the context of your assembly file, add the `--disam` switch:

```
annasim your_filename.asm --disasm your_filename.dasm -m your_memfile.mem
```

The disassembly file will look like the following:

```
[0003: 8602]      lli     r3 2           # load constant 2 -> r3
[0004: 77c0]      push    r7 r3          # push result
[0005: 4fff]

[0006: 4fc1]      pop     r7 r3          # load value from stack
[0007: 67c0]
[0008: 82d5]      lwi     r1 &_var_i     # load address of variable "i"
[0009: 9200]
[000a: 7640]      sw      r3 r1 0        # store variable "i" to data segment
```

5.2 *Running the Simulator*

There are three modes for the simulator.

5.2.1 Runner

The Runner is invoked with the `-r` switch. To assemble and run a program:

```
annasim your_filename.asm -r
```

You can trace execution of your program with the `-t` switch. This will show you each instruction as it executes, what register changed from the instruction, and what the stack looks like.

```
annasim your_filename.asm -r -t
```

The trace output will look like the following:

```
[0004: 0x493f] loop: addi    r4 r4 -1    |  r4: 0x0001
[0005: 0xa805]       beq     r4 &end     |
[0006: 0x0650]       add     r3 r1 r2    |  r3: 0x0008
[0007: 0x0280]       add     r1 r2 r0    |  r1: 0x0005
[0008: 0x04c0]       add     r2 r3 r0    |  r2: 0x0008
[0009: 0x3600]       out     r3          |
[000a: 0xa0f9]       beq     r0 &loop    |
```

```
[0004: 0x493f] loop: addi    r4 r4 -1     |  r4: 0x0000
[0005: 0xa805]       beq     r4 &end      |

[000b: 0x3000] end:  halt                 |
Halted at PC: 0x000b (42 cycles)
```

As each instruction is executed, you will see the PC, the instruction bits, the line of assembly code, and the new value of a register if one was changed. If your program uses R(7) as a stack pointer, it will also show the stack. This is a very handy way to debug a program, by getting a complete history of its execution.

5.2.2 Debugger

The debugger is a very minimal console debugger in the style of gdb. To assemble and invoke the debugger:

```
anna your_filename.asm -d
```

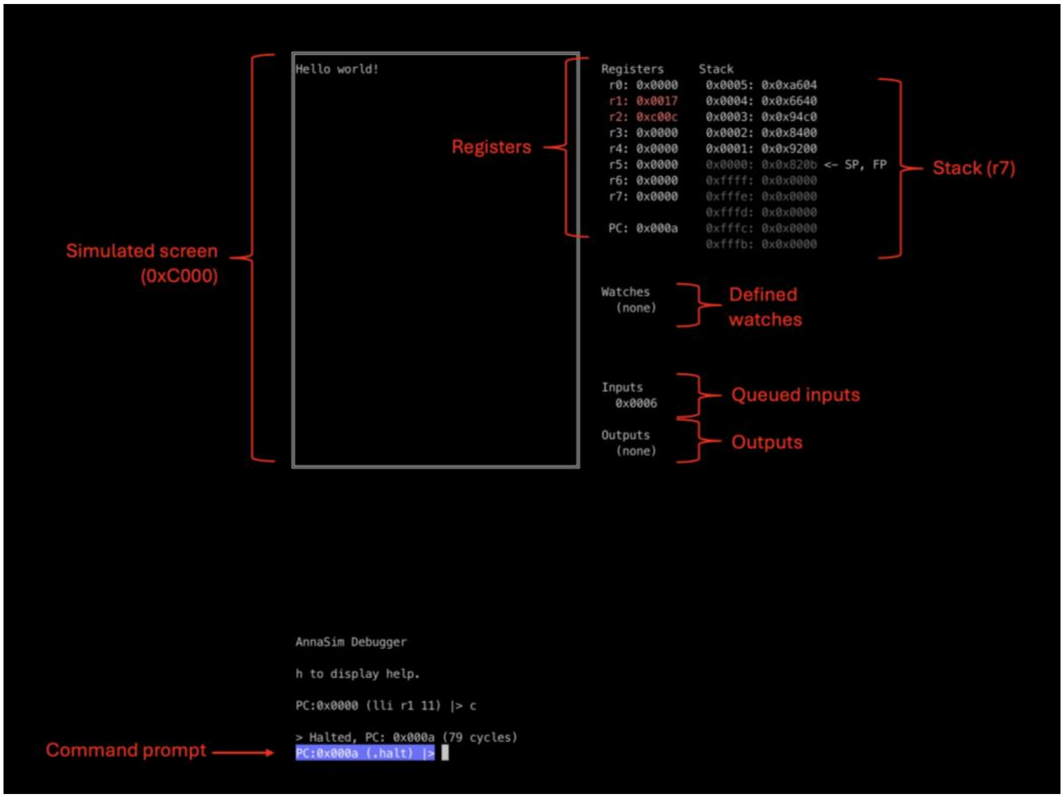
To see all the commands in the debugger, type h and press enter.

Common commands include:

h	Display help.
b <line>	Break at line during execution.
b <label>	Break at label during execution.
c	Continue execution until halted.
f <frame level>	Show the current stack frame. Can specify a frame level as a negative number, which follows the pointer at FP+0 to display the previous stack frame. For example, f -2 displays two frames back.
i	Add input values to input queue. Can specify multiple values at once.
m <address>	Display memory at address.
n	Execute next instruction. (Pressing enter on a blank line defaults to n.)
rn	Display contents of register N.
r*	Display contents of all registers.
w <address>	Add address to watches. After every execution, displays all memory contents at the watched addresses.
W	Clear all watches.

5.2.3 VT100 advanced debugger

The advanced debugger is an enhancement to the basic debugger that adds several displays that update as the program executes.



The same commands in the basic debugger work in the advanced debugger.

The advanced debugger includes a simulated memory-mapped screen. The screen is a 25 row, 40 column display mapped to memory addresses 0xC000 through 0xC3E7 in row order. Each word in this range is interpreted as an ASCII character code and displayed accordingly. For example, a value of 65 in address 0xC029 would display a capital-A in the second column of the second row of the screen.

The following program prints `Hello world!` to the first line of the screen:

```
screen: .def    0xc000

        lwi     r1 &msg
        lwi     r2 &screen

loop:   lw      r3 r1 0
        beq     r3 &done
        sw      r3 r2 0
        addi    r1 r1 1
```

```
        addi    r2 r2 1
        beq     r0 &loop

done:   halt

msg:    .cstr   "Hello, world!"
```

5.3 Inputs and outputs

To be able to write and run programs without input devices like keyboards, the ANNA CPU includes an input queue and the ability to output values.

5.3.1 Specifying inputs

The `in` instruction will take the next value from the input queue and place it into a register. You may specify an input using the `i` command in the debuggers, or by specifying inputs on the command line. The following command line runs a program with two inputs, 5 and 10, added to the input queue.

```
anna test.asm -d -i 5 -i 10
```

5.3.2 Outputs

The `out` instruction will cause the simulator to output a hexadecimal value from the specified register. How the value will be displayed depends on how the simulator was run.

- If run using the `-r` runner or `-d` debugger, the value will simply be printed.
- If run using the `-a` advanced debugger, the value will be added to the Outputs section of the debugger.

The `outs` instruction will print the NUL-terminated string pointed to by *Rd* to STDOUT.

The `outn` instruction will print the number in *Rd* as a decimal integer to STDOUT.

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.
- A "register map" that explains the use of each register.
- If certain registers have a single role, consider aliasing them with a more descriptive name using the `.ralias` directive.
- For each function, indicate what the code does and include a register map.

- Place a brief comment for each logical segment of code. Since assembly language programs are notoriously difficult to read, good comments are absolutely essential! o 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 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` and `.cstr` in the data section.