ANNA+ Programming Card

Opcode	***Op***	***Operands***	***Description***
0000	`add`	*Rd* Rs_1 Rs_2	Two's complement addition: R(*Rd*) ← R(Rs_1) + R(Rs_2)
0000	`sub`	*Rd* Rs_1 Rs_2	Two's complement subtraction: R(*Rd*) ← R(Rs_1) – R(Rs_2)
0000	`and`	*Rd* Rs_1 Rs_2	Bitwise and operation: R(*Rd*) ← R(Rs_1) & R(Rs_2)
0000	`or`	*Rd* Rs_1 Rs_2	Bitwise or operation: R(*Rd*) ← R(Rs_1) | R(Rs_2)
0000	`not`	*Rd* Rs_1	Bitwise not operation: R(*Rd*) ← ~R(Rs_1)
0000	`mul`	*Rd* Rs_1 Rs_2	Two's complement multiplication: R(*Rd*) ← R(Rs_1) × R(Rs_2)
0000	`div`	*Rd* Rs_1 Rs_2	Two's complement integer division: R(*Rd*) ← R(Rs_1) ÷ R(Rs_2)
0000	`mod`	*Rd* Rs_1 Rs_2	Two's complement modulus: R(*Rd*) ← R(Rs_1) % R(Rs_2)
0001	`jalr`	*Rd* Rs_1	Jumps to the address stored in register *Rd* and stores PC + 1 in register Rs_1.
0010	`in`	*Rd*	Input instruction: R(*Rd*) ← input
0011	`out`	*Rd*	Output instruction: output ← R(*Rd*). If *Rd* is `r0`, halts the processor (see `halt`).
0011	`outn`	*Rd*	Prints the integer value R(*Rd*) to STDOUT.
0011	`outs`	*Rd*	Prints the NUL-terminated string at M[R(*Rd*)] to STDOUT.
0100	`addi`	*Rd* Rs_1 *Imm6*	Add immediate: R(*Rd*) ← R(Rs_1) + *Imm6*
0101	`shf`	*Rd* Rs_1 *Imm6*	Bit shift. The contents of Rs_1 are shifted left (if *Imm6* is positive) or right with zero extension (if *Imm6* is negative). The shift amount is abs(*Imm6*); the result is stored in R(*Rd*).
0110	`lw`	*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*]
0111	`sw`	*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*)
1000	`lli`	*Rd Imm8*	The lower bits (7-0) of *Rd* are copied from *Imm8*. The upper bits (15-8) of *Rd* are equal to bit 7 of *Imm8* (sign extension).
1001	`lui`	*Rd Imm8*	The upper bits (15- 8) of *Rd* are copied from Imm8. The lower bits (7-0) of *Rd* are unchanged.
1010	`beq`	*Rd Imm8*	If R(*Rd*) = 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
1011	`bne`	*Rd Imm8*	If R(*Rd*) ≠ 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.

1100	`bgt`	*Rd Imm8*	If R(*Rd*) > 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
1101	`bge`	*Rd Imm8*	If R(*Rd*) ≥ 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
1110	`blt`	*Rd Imm8*	If R(*Rd*) < 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
1111	`ble`	*Rd Imm8*	If R(*Rd*) ≤ 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
Pseudo-Ops	`br`	*Imm8*	Assembles as `beq r0 Imm8` to always branch.
	`halt`		Assembles as `out r0` instruction (`0x3000`) that halts the processor.
	`jmp`	*Rd*	Assembles as `jalr Rd r0` to perform a jump.
	`lwi`	*Rd Imm16*	Assembles `lli` and `lui` instructions to load *Imm16* into R(*Rd*). Can be used with labels.
	`mov`	*Rd* Rs_1	Assembles as `add Rd Rs1 r0` to execute R(*Rd*) ← R(Rs_1)
	`push`	*Rsp* Rs_1	Assembles `sw` and `addi` instructions to push R(Rs_1) to M(*Rsp*) and decrement R(*Rsp*).
	`pop`	*Rsp Rd*	Assembles `addi` and `lw` instructions to increment R(*Rsp*) then pop M(*Rsp*) to R(*Rd*).
Assembler Directives	`.halt`		Assembler directive that emits an `out` instruction (`0x3000`) that halts the processor. (Supported for backward compatibility; use `halt` pseudo-op instead.)
	`.fill`	*Imm16*	Fills next memory locations with the specified values. Immediate is a signed value.
	`.frame`	*String*	Defines stack frames (activation records) for function. See the ANNA Guide for more information.
	`.org`	*Imm16*	Assembly continues at the address indicated.
	`.def`	*Imm16*	Sets the specified label to the value indicated. Must specify a label with this directive. Similar to `.fill` except that `.def` does not write any data to memory.
	`.cstr`	*String*	Fills next memory locations with a NUL-terminated string, one character per memory word.
	`.ralias`	*A* R_n	Creates an alias *A* for register *n*. The alias must start with an `r`.

Registers

- Represented by fields *Rd*, Rs_1, and Rs_2.

- 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.

Immediates

- Represented by fields *Imm6*, *Imm8*, and *Imm16*. The number refers to the size of the immediate in bits.
- Immediates are represented using decimal values, hexadecimal values, or labels. Hexadecimal values must start with '`0x`' and labels must be preceded with '`&`'.
- The immediate fields represent a signed 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.
- Labels refer to the address of the label. If a label is used in a branch, the proper PC-relative offset is computed and used as the immediate.

Comments

- A comment begins with a pound sign '`#`' and continues until the following newline.

Labels

- Label definitions consist of a string of letters, digits, and underscore characters followed by a colon. The colon is not part of the label name.
- A label definition must precede an instruction on the same line.
- A label may only be defined once in a program. Only one label is allowed per instruction. The instruction must appear on the same line as the label.

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*

*Function codes for opcode 0000: `add` (000), `sub` (001), `and` (010), `or` (011), `not` (100), `jalr` , `in`, `out` do not use the function; each has a unique opcode.

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*

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.

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
```