ANNA Programming Card

Opcode	***Op***	***Operands***	***Description***
0000	`add`	R_D R_A R_B	Two's complement addition: R(R_D) ← R(R_A) + R(R_B)
0000	`sub`	R_D R_A R_B	Two's complement subtraction: R(R_D) ← R(R_A) – R(R_B)
0000	`and`	R_D R_A R_B	Bitwise and operation: R(R_D) ← R(R_A) & R(R_B)
0000	`or`	R_D R_A R_B	Bitwise or operation: R(R_D) ← R(R_A) | R(R_B)
0000	`not`	R_D R_A	Bitwise not operation: R(R_D) ← ~R(R_A)
0001	`jalr`	R_D R_A	Jumps to the address stored in R_D and stores PC + 1 in R_A.
0010	`in`	R_D	Input instruction: R(R_D) ← input
0011	`out`	R_D	Output instruction: output ← R(R_D). If R_D is `r0`, halts the processor.
0100	`addi`	R_D R_A *Imm6*	Add immediate: R(R_D) ← R(R_A) + *Imm6*
0101	`shf`	R_D R_A *Imm6*	Bit shift. The contents of R_A 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_D.
0110	`lw`	R_D R_A *Imm6*	Loads word from memory; effective address is computed by adding the contents of R_A with the signed immediate: R(R_D) ← M[R(R_A) + *Imm6*]
0111	`sw`	R_D R_A *Imm6*	Stores word into memory; effective address is computed by adding the contents of R_A with the signed immediate: M[R(R_A) + *Imm6*] ← R(R_D)
1000	`lli`	R_D *Imm8*	The lower bits (7-0) of R_D are copied from *Imm8*. The upper bits (15-8) of R_D are equal to bit 7 of *Imm8* (sign extension).
1001	`lui`	R_D *Imm8*	The upper bits (15- 8) of R_D are copied from Imm8. The lower bits (7-0) of R_D are unchanged.
1010	`beq`	R_D *Imm8*	If R(R_D) = 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
1011	`bne`	R_D *Imm8*	If R(R_D) ≠ 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
1100	`bgt`	R_D *Imm8*	If R(R_D) > 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
1101	`bge`	R_D *Imm8*	If R(R_D) ≥ 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
1110	`blt`	R_D *Imm8*	If R(R_D) < 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
1111	`ble`	R_D *Imm8*	If R(R_D) ≤ 0, then branch is taken with indirect target of PC + 1 + *Imm8* as next PC. Immediate is a signed value.
Assembler Directives	`.halt`		Assemble directive that emits an `out` instruction (`0x3000`) that halts the processor.
	`.fill`	*Imm16*	Assembler directive that fills next memory location with the specified value. Immediate is a signed value.

Registers

- Represented by R_D, R_A, and R_B.
- 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 *Imm6*, *Imm8*, and *Imm16*. The number refers to the size of the immediate in bits.
- Immediates can be specified using decimal values, hexadecimal values, or labels. Hexadecimal values must start with '`0x`' and labels must be preceded with '`&`'.
- Immediates represent a signed value. The immediate 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	R_D	R_A	R_B	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 code; each has a unique opcode.

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

15 12	11 9	8 6	5 0
Opcode	R_D	R_A	*Imm6*

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

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

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

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
```