### RASM:

RASM differs from OVT in that it has more instructions but more importantly it has 4 bytes per instruction.

The function of the bytes is as followed:

1. Instruction byte (opcode)
2. Source byte for the first operand (src1)
3. Source byte for the second operand (scr2)
4. Destination for where the result should be stored (dst)

This Regi also has more complex memory so a few notation differences:

|  |  |  |
| --- | --- | --- |
|  | operand | Result |
| a | The value a | Store at a |
| <a> | The value at register a | Store at the value in register a. (store at <a>) |
| [a] | Option to choose between a and <a> | Option to choose between a and <a> |

Some instructions start with a, b and/or c (i.e., abc01000). This signifies where the inputs come from and where the result is stored.

|  |  |  |
| --- | --- | --- |
| a | =0 | First operand is from the register where src1 points. (<src1>) |
|  | =1 | First operand is an immediate with value src1. |
| b | =0 | Second operand is from the register where src1 points. (<src2>) |
|  | =1 | Second operand is an immediate with value src1. |
| c | =0 | Result is saved at the register that is given by register dst. (<<dst>>) |
|  | =1 | Result is saved at the register dst. (<dst>) |

-> [z]: store at [z]

Similar to OVT, keywords aren’t case sensitive, but variables assigned with const or label are.

The instructions are divided into four categories:

1. xxx00xxx: loops & functions (& halt/exit)
2. xxx01xxx: (bitwise) logic
3. xxx10xxx: arithmetic
4. xxx11xxx: bit shifts & stack

A dash means that that bit doesn’t have any effect on the instruction, and ‘x’ and ‘y’ are numbers that control what the instruction does.

loops & functions

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| JumpGt [x] [y] [z] | If x > y, jump to z (greater than) | abc0 0001 |
| JumpEq [x] [y] [z] | If x = y, jump to z (equal) | abc0 0010 |
| JumpGE [x] [y] [z] | If x >= y, jump to z (greater equal) | abc0 0011 |
| JumpLt [x] [y] [z] | If x = y, jump to z (less than) | abc0 0100 |
| JumpNE [x] [y] [z] | If x ≠ y, jump to z (not equal) | abc0 0101 |
| JumpLE [x] [y] [z] | If x <= y, jump to z (less equal) | abc0 0110 |
| Jump [z] | Jump to z | a0c0 0111 |

(nog niet toegevoegd: Call en icall lezen bij a=1, twee data bytes zodat je ook functions kan hebben na line 256) (ook niet toegevoegd: jump leest bij a=1, twee data bytes voor het jumpen naar lines na line 256)

(bitwise) logic

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| and [x] [y] [z] | x&y->z | abc0 1000 |
| or [x] [y] [z] | x|y->z | abc0 1001 |
| nand [x] [y] [z] | ¬(x&y)->z | abc0 1010 |
| nor [x] [y] [z] | ¬(x|y)->z | abc0 1011 |
| xor [x] [y] [z] | x^y->z | abc0 1100 |
| xnor [x] [y] [z] | ¬(x^y)->z | abc0 1101 |
| test [x] [y] [z] | Checks if the bit with index y in x is a 1. If it is a one: 1->z, else 0->z | abc0 1110 |
| not [x] [z] | ¬x->z | a0c0 1111 |
| high [x] [z] | Stores the index of the highest bit in z. If there is no 1 then it stores 16 (0x10000). | a1c0 1111 |

arithmetic

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| add [x] [y] [z] | x+y->z (set carry flag when overflowed) | abc1 0000 |
| addc [x] [y] [z] | Add x, y and the carry flag then store in z (set carry flag when overflowed) | abc1 0001 |
| Sub [x] [y] [z] | x-y->z (set carry flag when underflowed) | abc1 0010 |
| subb [x] [y] [z] | Subtract y and carry flag from x then store in z (set carry flag when underflowed) | abc1 0011 |
| mult [x] [y] [z] | x\*y->z (overflow -> carry register) | abc1 0100 |
| div [x] [y] [z] | x/y->z (x mod y -> carry register) | abc1 0101 |
| inc [x] [z] | x+1->z | a-c1 0110 |
| dec [x] [z] | x-1->z | a-c1 0111 |

Bit shift & stack

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| shl [x] [y] [z] | Shift x a total of y places to the left->z (0<=y<=16)(set carry register) | abc1 1000 |
| shlc [x] [y] [z] | shift x a total of y places to the left and fill from the right from carry ->z (0<=y<=16)(set carry register) | abc1 1001 |
| shr [x] [y] [z] | Shift x a total of y places to the right->z (0<=y<=16)(set carry register) | abc1 1010 |
| shrc [x] [y] [z] | shift x a total of y places to the right and fill from the left from carry ->z (0<=y<=16)(set carry register) | abc1 1011 |
| Rotl [x] [y] [z] | Rotate x a total of y places to the left -> z | abc1 1100 |
| Rotr [x] [y] [z] | Rotate x a total of y places to the right -> z | abc1 1101 |
| copy [x] [z] | x -> z | a-c1 0000 |
| Push [x] | Increment stack pointer and then push x to stack | a001 1111 |
| Pop [x] | Pop from stack to x and then decrement stack pointer | a011 1111 |
| Exec [x] | Execute reg(x)||reg(x+1) | a101 1111 |

The stack grows down from memory address 0 and shares the memory with all other operations.

Conditionals (only after RASM v2.1)

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| uJumpGt [x] [y] [z] | Unsigned If x > y, jump to z (greater than) | 1 abc0 0001 |
| uJumpEq [x] [y] [z] | Unsigned If x = y, jump to z (equal) | 1 abc0 0010 |
| uJumpGE [x] [y] [z] | Unsigned If x >= y, jump to z (greater equal) | 1 abc0 0011 |
| uJumpLt [x] [y] [z] | Unsigned If x = y, jump to z (less than) | 1 abc0 0100 |
| uJumpNE [x] [y] [z] | Unsigned If x ≠ y, jump to z (not equal) | 1 abc0 0101 |
| uJumpLE [x] [y] [z] | Unsigned If x <= y, jump to z (less equal) | 1 abc0 0110 |
| uJump [x] [z] | Unsigned Jump to x||z | 1 a0c0 0111 |

Floats arithmetic (RASM v2.2)

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| Fadd [x] [y] [z] | x+y->z (set carry flag when overflowed) (float) | 10 abc1 0000 |
| FSub [x] [y] [z] | x-y->z (set carry flag when underflowed) (float) | 10 abc1 0001 |
| Fmult [x] [y] [z] | x\*y->z (overflow -> carry register) (float) | 10 abc1 0010 |
| Fdiv [x] [y] [z] | x/y->z (x mod y -> carry register) (float) | 10 abc1 0011 |
| Ftoint [x] [z] | Cast a float to integer (int(x)-> z) | 10 abc1 0100 |
| Inttof [x] [z] | Cast integer to float (float(x) -> z) | 10 abc1 0101 |

Floats conditionals (RASM v2.2)

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| FJumpGt [x] [y] [z] | (float) If x > y, jump to z (greater than) | 10 abc0 0001 |
| FJumpEq [x] [y] [z] | (float) If x = y, jump to z (equal) | 10 abc0 0010 |
| FJumpGE [x] [y] [z] | (float) If x >= y, jump to z (greater equal) | 10 abc0 0011 |
| FJumpLt [x] [y] [z] | (float) If x = y, jump to z (less than) | 10 abc0 0100 |
| FJumpNE [x] [y] [z] | (float) If x ≠ y, jump to z (not equal) | 10 abc0 0101 |
| FJumpLE [x] [y] [z] | (float) If x <= y, jump to z (less equal) | 10 abc0 0110 |
| FJump [x] [z] | (float) Jump to x||z | 10 a0c0 0111 |
| Call [x] [y] [z] | Go to function [x][[1]](#footnote-1), if conditional is true (uvw follows the last three bits of jump conditionals) and sets the subroutine register | 100 abc0 0uvw |
| Ret [x] [y] [z] | Return to where call/icall was last used, when conditional is true (uvw follows the last three bits of jump conditionals) | 101 0010 0uvw  (or 101 0000 0uvw???) |
| Exit | Exit program | 110 ---- ---0 |
| Hlt | Halt until next interrupt | 110 ---- ---1 |
| CopyPtr [x] [y] | Copy <x> y | 111 a-c1 0000 |
| ICall [x] | Go to function [x] if there has been an interrupt since last time icall/iret was used | 1100 abc0 0uvw |
| IRet | Return to where call/icall was last used if there has been an interrupt since last time icall/iret was used | 1101 0010 0uvw |

Due to how the computer is wired it is possible to make every comparison depend on the interrupt but that isn’t implemented in the compiler

Registers:

* reg0-31: registers
* Reg32: carry
* Reg33-35: input/output
* Reg36-37: output
* reg32768 (0x8000): (read only) ram or (write only) set ram address (v2.1)
* reg 40960-49151 (0xA000-0xBFFF): write to specified address in ram (v2.1)
* Reg49152 (0xC000): (write only) counter (v2.1)

Const <name> <value>: Every .ovt program can start with a few lines that start with ‘const’.

assigns the value <value> to <name>. From then on you use <name> instead of <value>.

Label <name>: assigns the line number of the next instruction to <name>. This is useful for loops.

; (comments): everything after ‘;’ is seen as a comment (until a new line).

1. Call workings: call [a], jump to line a, at line a is a jump statement that brings jumps to function a. The instruction you call at is saved so you can later use ‘ret’ to return. This is saved by pushing it to the stack. [↑](#footnote-ref-1)