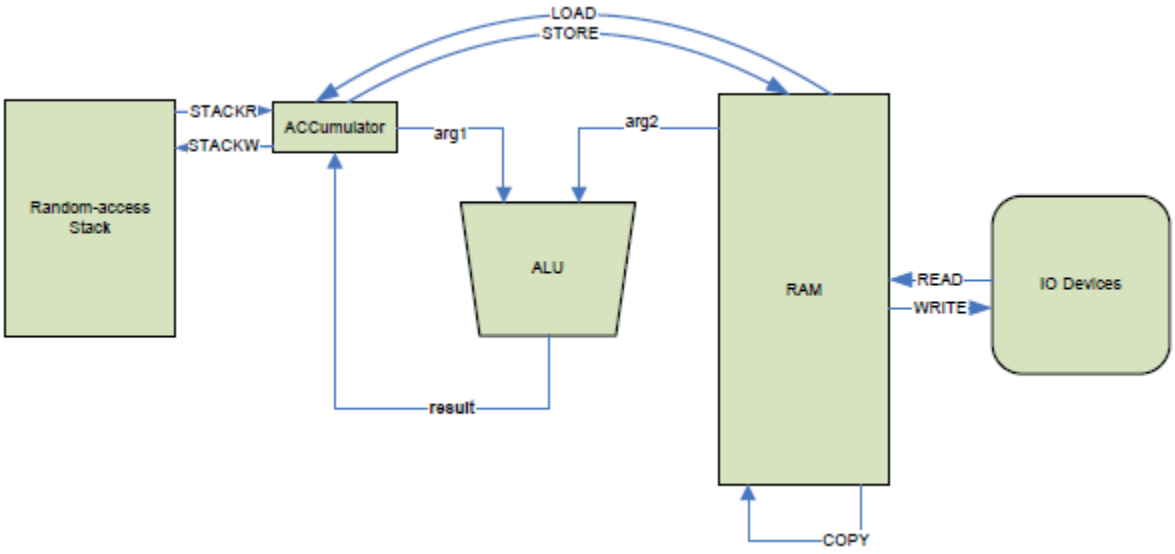
Simple ACC Assembler

Assembly language is machine dependent, depending on a specific computer architecture.

Assembly language can be translated into lower level machine language to execute on a machine, or be interpreted in software, leading to virtual machine.

This virtual computer is a simple computer with

- ALU, RAM, just one register called ACCumulator, simple IO, and a built-in stack
- Word size is 2 bytes, and the addressability is thus 64k
- The only type is signed integer, with range [-32k,+32k]



Program Format

- Each line
  - independent and self-contained containing
    - blank
    - a complete instruction
    - a complete storage directive
  - all delimiters are WS
- Instruction
  - for the accumulator machine
    - left argument and result are in an implicit accumulator ACC register, except for COPY)
    - with the following format
      [Label:] XXX arguments
      - Label is optional

        - XXX is the reserved name in upper case
      - arguments as needed separated by spaces
- Storage directive
  - yyy val
    - yyy is storage (variable) name
    - val is the initial value
    - all storage are signed 2 byte integers
    - all storage is global in scope and lifetime
- Names
  - Instruction names are reserved in upper case
  - Variables start will letter and continue with letters and digits up to 8 length
- Numbers
  - The computer uses standard data 2's complement data representation so thus data range is -32k to +32k.

Reserved Instruction Set (# arguments, semantics)

- BR (1, jump to arg)
- BRNEG (1, jump to arg if ACC <0)
- BRZNEG (1, jump to arg if ACC <=0)
- BRPOS (1, jump to arg if ACC >0)
- BRZPOS (1, jump to arg if ACC >=0)
- BRZERO (1, jump to arg if ACC ==0)
- COPY (2, arg1 = arg2)
- ADD (1, ACC = ACC +arg
- SUB (1, ACC = ACC - arg)
- DIV (1, ACC = ACC / arg)
- MULT (1, ACC = ACC * arg)
- READ (1, arg=input integer)
- WRITE (1, put arg to output as integer)
- STOP (0, stop program)
- STORE (1, arg = ACC)
- LOAD (1, ACC=arg)
- NOOP (0, nothing)
- Stack operations as seen below

Immediate Values

- ADD, DIV, MULT, WRITE, LOAD, SUB
  - can take either variable or immediate value as the argument
  - immediate value is positive integer or negative 2-byte integer

Stack

- PUSH (0, tos++)

- POP (0, tos--)
- STACKW (1,stack[tos-arg]=ACC)
- STACKR (1,ACC=stack[tos-arg])

PUSH/POP

- means to reserve/delete automatic storage

STACKW/STACKR n

- stack random access instructions
- n must be a non-negative integer
- the access is to n-th element down from tos (top of the stack)
  NOTE: tos points to the topmost element on the stack at offset n=0

Semantics

Execution begins with the first instruction, continues sequentially unless one of the BRs is executed, until STOP is reached.

All storage directive variables are persistent. All data 2 bytes signed integers.

Invocation

> `VirtMach` // read from stdin

> `VirtMach file.asm` // read from file.asm

Note that .asm extension is not required as it is not implicit.

Location

`/accounts/classes/janikowc/cs4280/asmInterpreter/VirtMach`

- readable executable, copy to your directory

IDE for the interpreter, with debugging and step execution

- https://comp.umsl.edu/assembler

Examples

Simple flows

`Add1.asm`. Read a number and print number+1

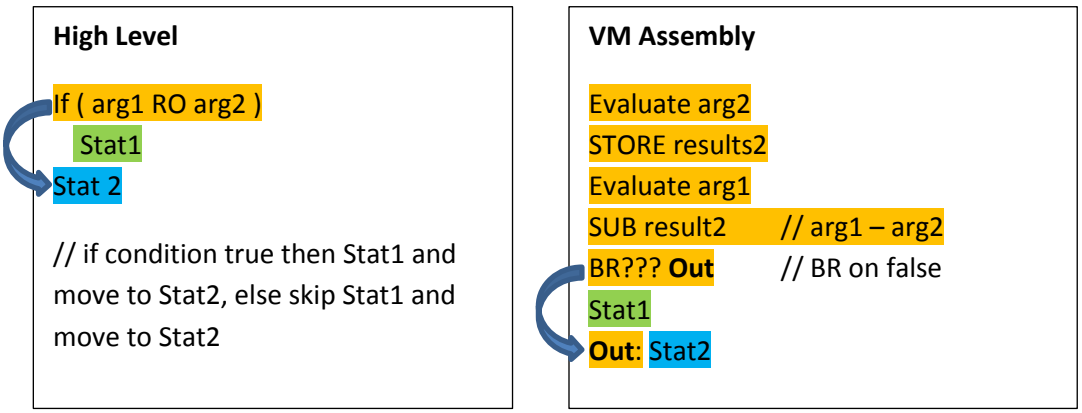
```
READ X
LOAD X
ADD 1
STORE X
WRITE X
STOP
X 0
```

`SumOf3.asm`. Read 3 numbers, add them and display the total, using 3 variables

```
READ X
READ Y
READ Z
LOAD X
ADD Y
ADD Z
STORE X
WRITE X
STOP
X 0
Y 0
Z 0
```

Conditional Flow

- Simple relational conditions are processed by subtracting sides and implementing proper jump
- This is 'if' without else



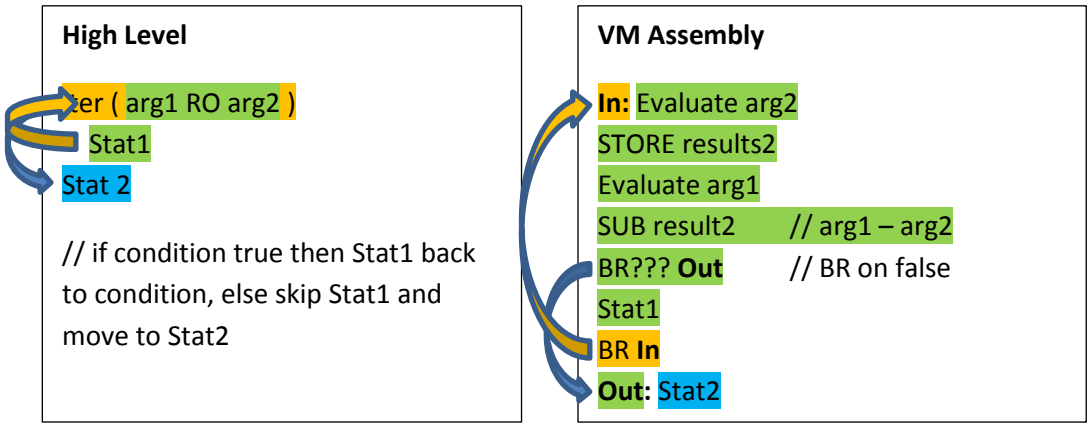
`If.asm`. Read a number and print it if >= 1

```
int  x
scanf x
if ( x >= 1 )
  print x .
```

```
READ x
LOAD x
SUB 1
BRNEG Out
WRITE x
Out: STOP
x  0
```

Iteration

- Iteration is similar to conditional except that upon executing the 'true' statement there is unconditional jump back to evaluate the condition again.
- This will continue (iterate) until the condition is false



`Loop.asm`. Read input number and print number then number – 1 … down to 2

```
int x
read x
while ( x >= 2  )
{    print x
     x = x - 1
}
```

```
READ x
In: LOAD x
SUB 2
BRNEG Out
WRITE x
LOAD x
```

```
SUB 1
STORE x
BR In
Out: STOP
x 0
```

Stack

The stack uses PUSH to advance the TOS (top of stack), which is just storage reservation (no data placed on the stack). The stack also uses POP to reduce the TOS, which is deallocating storage on the top (no data read off the stack).

The stack also uses STACKW/STACKR to write/read the stack at any location on the stack. The location to write/read is specified by the argument, which is offset from the TOS so that data at TOS has offset 0, one below offset 1, etc. For example, to read the value off the TOS you would do

```
STACKR 0
```

and to delete that space on TOS you would do

```
POP
```

Example

`Stack1.asm`. In this example, 2 numbers are read in, added, and the sum is displayed back. Only one variable is used, X. The first read-in number goes to X (all inputs goes to variables), then it is placed on the stack before the second number is read into the same X (overwriting the first value). The first value could be saved just in ACC but in this example stack is used. To save it, first PUSH allocates new storage on the stack, then the first value is loaded from X to ACC, and then written on the stack with STACKW 0 - value must be in ACC to be written to the stack.

After that, the second number is read into X, the stack saved value is retrieved from the stack to ACC using STACKR 0, the stack storage is removed using POP, the addition is performed and the number is saved back to X to be printed out.

Both STACKR and STACKW use 0 as the only value on the stack is on the TOS and thus the offset from TOS is always 0.

```
READ X
PUSH
LOAD X
STACKW 0
READ X
STACKR 0
POP
ADD X
STORE X
```

```
WRITE X
STOP
X 0
```

Example

`Stack2.asm`. Read first argument, and then read as many arguments as the first argument stated and returns the sum. All integers.

```
      READ X
      COPY Z X
LOOP1: LOAD X
      BRZERO OUT1
      BRNEG  OUT1
      READ Y
      LOAD Y
      PUSH
      STACKW 0
      LOAD X
      SUB 1
      STORE X
      BR LOOP1
OUT1:  NOOP
      LOAD 0
      STORE Y
LOOP2: STACKR 0
      ADD Y
      STORE Y
      POP
      LOAD Z
      SUB 1
      BRZERO OUT2
      BRNEG OUT2
      STORE Z
      BR LOOP2
OUT2:  NOOP
      WRITE Y
STOP
X 0
Y 0
Z 0
```