Accumulator machine

We will use the accumulator machine architecture to demonstrate pass1 and pass2.

The accumulator machine
- has one processor register: the accumulator
- all other operands are in memory, and expressions require a sequence of load/operate/store instructions.
The image contains a table listing instructions for an accumulator machine. The table is as follows:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Address</th>
<th>Operation Name</th>
<th>Machine Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>----</td>
<td>halt</td>
<td>stop execution</td>
</tr>
<tr>
<td>div</td>
<td>addr</td>
<td>divide</td>
<td>acc = acc/memory[addr]</td>
</tr>
<tr>
<td>mul</td>
<td>addr</td>
<td>multiply</td>
<td>acc = acc*memory[addr]</td>
</tr>
<tr>
<td>sub</td>
<td>addr</td>
<td>subtract</td>
<td>acc = acc-memory[addr]</td>
</tr>
<tr>
<td>add</td>
<td>addr</td>
<td>add</td>
<td>acc = acc+memory[addr]</td>
</tr>
<tr>
<td>load</td>
<td>addr</td>
<td>load</td>
<td>acc = memory[addr]</td>
</tr>
<tr>
<td>store</td>
<td>addr</td>
<td>store</td>
<td>memory[addr] = acc</td>
</tr>
<tr>
<td>ba</td>
<td>addr</td>
<td>branch always</td>
<td>pc = addr</td>
</tr>
<tr>
<td>blt0</td>
<td>addr</td>
<td>branch on less than</td>
<td>if acc&lt;0 then pc = addr</td>
</tr>
<tr>
<td>ble0</td>
<td>addr</td>
<td>branch on less than or equal</td>
<td>if acc&lt;=0 then pc = addr</td>
</tr>
<tr>
<td>beq0</td>
<td>addr</td>
<td>branch on equal</td>
<td>if acc==0 then pc = addr</td>
</tr>
<tr>
<td>bne0</td>
<td>addr</td>
<td>branch on not equal</td>
<td>if acc/=0 then pc = addr</td>
</tr>
<tr>
<td>bge0</td>
<td>addr</td>
<td>branch on greater than or equal</td>
<td>if acc&gt;=0 then pc = addr</td>
</tr>
<tr>
<td>bgt0</td>
<td>addr</td>
<td>branch on greater than</td>
<td>if acc&gt;0 then pc = addr</td>
</tr>
<tr>
<td>print</td>
<td>addr</td>
<td>print</td>
<td>display contents of memory[addr]</td>
</tr>
<tr>
<td>name</td>
<td>arg1</td>
<td>arg2</td>
<td>meaning</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>comment</td>
<td>text</td>
<td></td>
<td>allows for comments within program</td>
</tr>
<tr>
<td>end</td>
<td>symbol</td>
<td></td>
<td>defines starting address of program (should appear once, as the last line of the program)</td>
</tr>
<tr>
<td>label</td>
<td>symbol</td>
<td></td>
<td>defines a symbolic address within the program</td>
</tr>
<tr>
<td>word</td>
<td>symbol</td>
<td>value</td>
<td>defines a symbol address and allocates a data word at that address with the given value</td>
</tr>
</tbody>
</table>
Accumulator machine program - example 1

comment(` first example accumulator machine program ')
comment(` '')
comment(` data section for program -- word(label,value) '')
  word(a, 23)
  word(b, 45)
  word(c, 17)
  word(d, 0)
comment(` code that implements the expression d = a + b - c;')
label(start)
  load(a) comment(` ACC <- memory[a] ')
  add(b) comment(` ACC <- ACC + memory[b] ')
  sub(c) comment(` ACC <- ACC - memory[c] ')
  store(d) comment(` memory[d] <- ACC ')
  halt
comment(` start execution at label start ')
end(start)
Accumulator machine

- executable is in numeric format -- example assembly (numbers are in decimal rather than binary)
- the assembler has a location counter variable called "loc" that it uses in the first pass to assign addresses
accumulator machine

instruction table (built into assembler)

<table>
<thead>
<tr>
<th>name</th>
<th>opcode</th>
<th>args</th>
<th>loc increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>40</td>
<td>$1</td>
<td>2</td>
</tr>
<tr>
<td>sub</td>
<td>30</td>
<td>$1</td>
<td>2</td>
</tr>
<tr>
<td>load</td>
<td>50</td>
<td>$1</td>
<td>2</td>
</tr>
<tr>
<td>store</td>
<td>60</td>
<td>$1</td>
<td>2</td>
</tr>
<tr>
<td>halt</td>
<td>00</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Pass 1

first pass input -- symbol table after first pass

<table>
<thead>
<tr>
<th>assembly stmts</th>
<th>label</th>
<th>addr = loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>word(a,23)</td>
<td>a</td>
<td>00</td>
</tr>
<tr>
<td>word(b,45)</td>
<td>b</td>
<td>01</td>
</tr>
<tr>
<td>word(c,17)</td>
<td>c</td>
<td>02</td>
</tr>
<tr>
<td>word(d,0)</td>
<td>d</td>
<td>03</td>
</tr>
<tr>
<td>label(start)</td>
<td>start</td>
<td>04</td>
</tr>
</tbody>
</table>

...<< no other statements define labels >>...

...
Pass 2
second pass input "executable" after second pass
(loc:) assembly language addr: machine code or data
(00:) word(a,23) 00: 23 <- value in location a
(01:) word(b,45) 01: 45 <- value in location b
(02:) word(c,17) 02: 17 <- value in location c
(03:) word(d,0) 03: 0 <- value in location d
label(start)
(04:) load(b) 04: 50 <- opcode for load
       05: 00 <- address of a
(06:) add(c) 06: 40 <- opcode for add
       07: 01 <- address of b
(08:) sub(d) 08: 30 <- opcode for sub
       09: 02 <- address of c
(10:) store(a) 10: 60 <- opcode for store
       11: 03 <- address of d
(12:) halt 12: 00 <- opcode for halt
(13:) end(start) 13: 04 <- starting address
Control structures in assembly language

```c
int x;
int y;
if(x == 0)
    y = 1;
/* assume values for x, y, and one */
```

<table>
<thead>
<tr>
<th>C Source Code</th>
<th>Inefficient Assembly</th>
<th>Efficient Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x; int y; int(x == 0) y = 1;</td>
<td>load(x) beq0 process ba do_nothing</td>
<td>load(x) bne0 do_nothing</td>
</tr>
<tr>
<td></td>
<td>label(process) load(one) store(y)</td>
<td>label(process) load(one) store(y)</td>
</tr>
<tr>
<td></td>
<td>label(do_nothing) ...</td>
<td>label(do_nothing) ...</td>
</tr>
</tbody>
</table>
C if statement in accumulator machine code

```c
if(((x+b)>z))
    x+=y;
```

Accumulator machine code:

```c
/* assume x, b, y, z have values*/
load(x)
add(b)
sub(z)
ble0(do_nothing)  comment(` branch to do_nothing`)  
load(x)
add(y)
store(x)
label(do_nothing)
...```
C if-else statement in accumulator machine code

if(i == j)
    f=g+h;
else
    f=g-h;

Accumulator code:
/* assume values for f, i, j, g, and h */
load(i)
sub(j)
beq0(then_part)
label(else_part)
load(g)
sub(h)
ba(done)
label(then_part)
load(g)
add(h)
label(done)
store(f)
Loops

Implementing Loops

- All for loops, while loops, and do-while loops have an implicit branch from the bottom to the top of the loop.
- This branch instruction becomes explicit when translated into assembly.

```plaintext
while (x <= 10)
{
    load(x)
    x = x + y;
    label(loop)
    sub(ten)
    bgt0(done)
    load(x)
    add(y)
    store(x)
    ba(loop)
    label(done)
    test condition; exit loop if false
    body of loop; also updates the lcv
}
```
Alternate implementation: (eliminates one of the branch instructions in the loop)

```
while ( x <= 10 ) {
    x = x + y;
}
```

```
load(x)  
sub(ten)  
bgt0(done)
```

- test condition; skip loop if false

```
label(loop)
load(x)  
add(y)  
store(x)  
sub(ten)  
ble0(loop)
```

- body of loop; also updates the lcv
- test condition; loop again if true

```
label(done)
```
Loops in C/Assembly (accumulator machine code)

for loop
for ( x = 1; x <= y; x++ )
{
    z *= x;
}

rewritten as while loop
x = 1;
while ( x <= y )
{
    z *= x;
    x++; 
}
Loops in C/Assembly (accumulator machine code)

for loop - alternate implementation
(eliminates one of the branches in the loop)

for ( x = 1; x <= y; x++ )
{
    z *= x;
}

rewritten as while loop
x = 1;
while ( x <= y )
{
    z *= x;
    x++;
}
accumulator machine program—example 2

comment(`second example accumulator machine program `)
comment(`code that implements the loop `)
comment(`sum = 0; `)
comment(`for( i = 10; i > 0; i-- ){ `)
comment(`sum = sum + i; `)
comment(` } `)
label(begin)
  load(zero) comment(`sum = 0 ACC <- memory[zero] `)
  store(sum) comment(`memory[sum] <- ACC `)
  load(ten) comment(`i = 10 `)
  store(i)
label(loop)
  load(i) comment(`branch to done if i <= 0 `)
  ble0(done)
accumulator machine program – example 2 (cont’d)

load(sum) comment(` sum = sum + i '
   add(i)
   store(sum)
load(i)    comment(` i = i - 1 '
   sub(one)
   store(i)
   ba(loop)    comment(` unconditionally go back to loop ')
label(done)
halt
comment(` data section for program - can be at bottom ')
   word(sum,0)
   word(i,0)
   word(zero,0)
   word(one,1)
   word(ten,10)
comment(` start execution at label begin ')
end(begin)
Graphical simulator for the accumulator machine

CPU
  PC
  IR
  ACC

BIU
  MAR  ----  address lines --
  MDR  ----  data lines ------
          ----  control lines ---

Memory
Graphical simulator for the accumulator machine

CPU

- PC (program counter) - points to next instruction
- IR (instruction register) - holds current instruction

BIU (bus interface unit)

- MAR (memory address register) – a CPU register that either stores the memory address from which data will be fetched to the CPU or the address to which data will be sent and stored
- MDR (memory data register – a CPU register that contains the data to be stored in memory or the data after a fetch from memory.
Graphical simulator for the accumulator machine

1. copy the files into your directory as instructed in the lab
2. run the shell script “./assemble.sh” (which input from the specified file and produces numeric output in the file "executable")
3. run simulator “./acc” (reads program in numeric form from "executable")
   - you can click on "RT advance" to see the effect of one register transfer or "Instruction Advance" to see the effect of one instruction
   - "RT Run" will run the program, RT-step by RT-step (you control the interval between step by using the slider labeled "RT Run interval")
   - "Stop" stops the RT-run mode
   - "Reset" reloads the program from the file named" executable"
   - "Quit" exits the simulator