IBCM

C++ to assembly to machine code

hello.cpp

```
#include <iostream>
int main() {
  std::cout << "Hello!\n";
  return 0;
}
```

hello.o (not shown)

C++ to assembly to machine code

hello.cpp

```
#include <iostream>
int main() {
  std::cout << "Hello!\n";
  return 0;
}
```

hello.o (not shown)

hello.s

```
.LC0:
  .string "Hello!\n"
main:
  sub rsp, 8
  mov esi, .LC0        # arg1 ← "Hello!\n"
  mov edi, _ZSt4cout   # arg2 ← cout
  call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_P
                       # call operator<<
  xor eax, eax         # return value ← 0
  pop rdx
  ret                  # return
```

C++ to assembly to machine code

hello.cpp

```
#include <iostream>
int main() {
  std::cout << "Hello!\n";
  return 0;
}
```

hello.o (not shown)

hello.s

```
.LC0:
  .string "Hello!\n"
main:
  sub rsp, 8
  mov esi, .LC0        # arg1 ← "Hello!\n"
  mov edi, _ZSt4cout   # arg2 ← cout
  call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_P
                       # call operator<<
  xor eax, eax         # return value ← 0
  pop rdx
  ret                  # return
```

C++ to assembly to machine code

hello.cpp

```
#include <iostream>
int main() {
  std::cout << "Hello!\n";
  return 0;
}
```

hello.s

```
.LC0:
  .string "Hello!\n"
main:
  sub rsp, 8
  mov esi, .LC0        # arg1 ← "Hello!\n"
  mov edi, _ZSt4cout   # arg2 ← cout
  call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_P
                       # call operator<<
  xor eax, eax         # return value ← 0
  pop rdx
  ret                  # return
```

hello.o (not shown)

+ standard library

hello.exe

(actually binary bytes without comments,
but shown as hexadecimal bytes with comments)

```
…
48 83 ec 08       # sub rsp, 8
be 64 07 40 00    # mov esi, .LC0 == 0x400764
bf 60 10 60 00    # mov edi, _ZSt4cout == 0x601060
e8 bd ff ff ff    # call operator<<
31 c0             # xor eax, eax
5a                # pop rdx
c3                # ret
…
```

assembly language and machine language

machine language — what the physical hardware expects

- how it reads bytes of memories when looking for work

assembly language — text representation of that

- *direct* translation to machine code

why learn assembly?

designing hardware

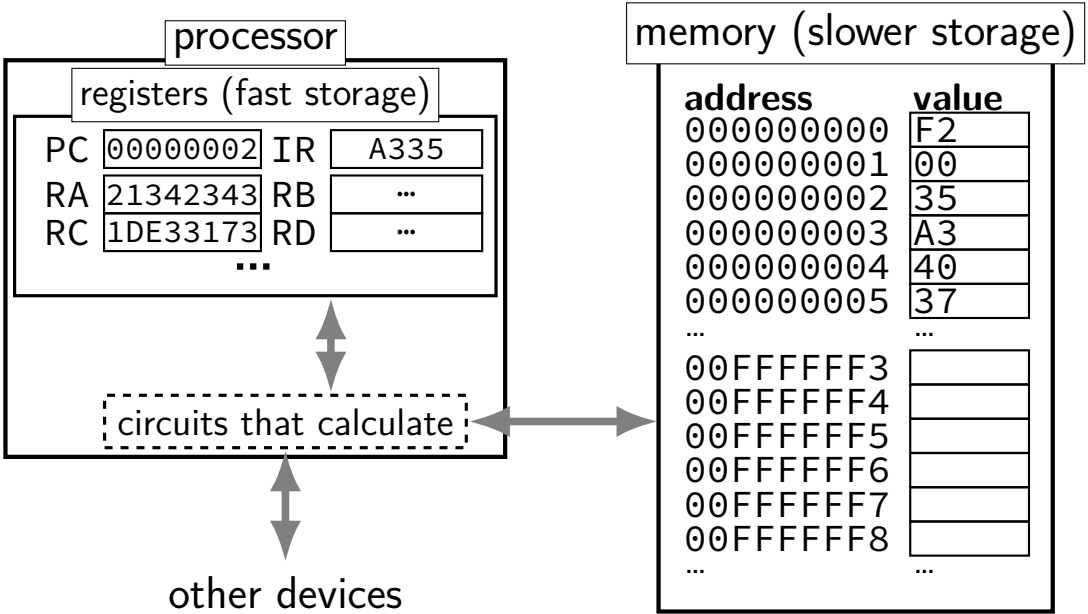
writing compilers

writing operating systems

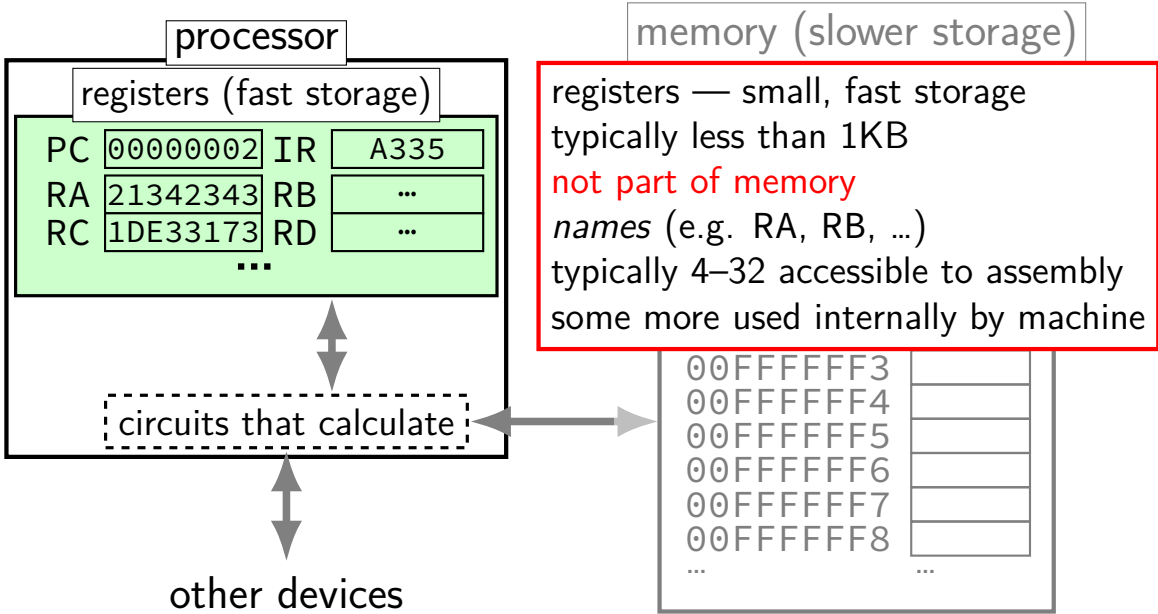
understanding how compilers work

understanding how computers work

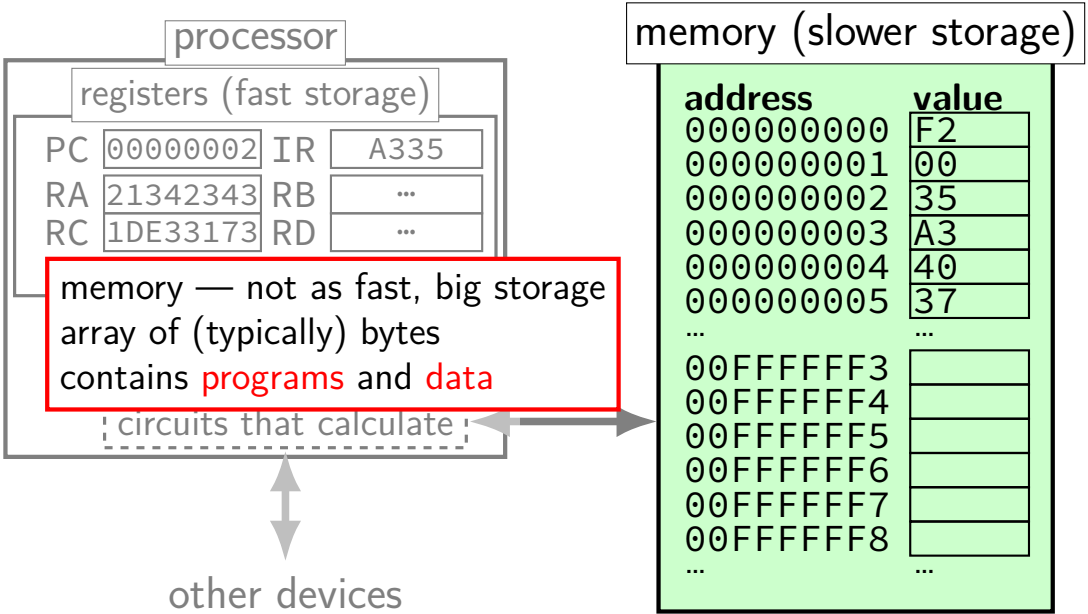
our machine model



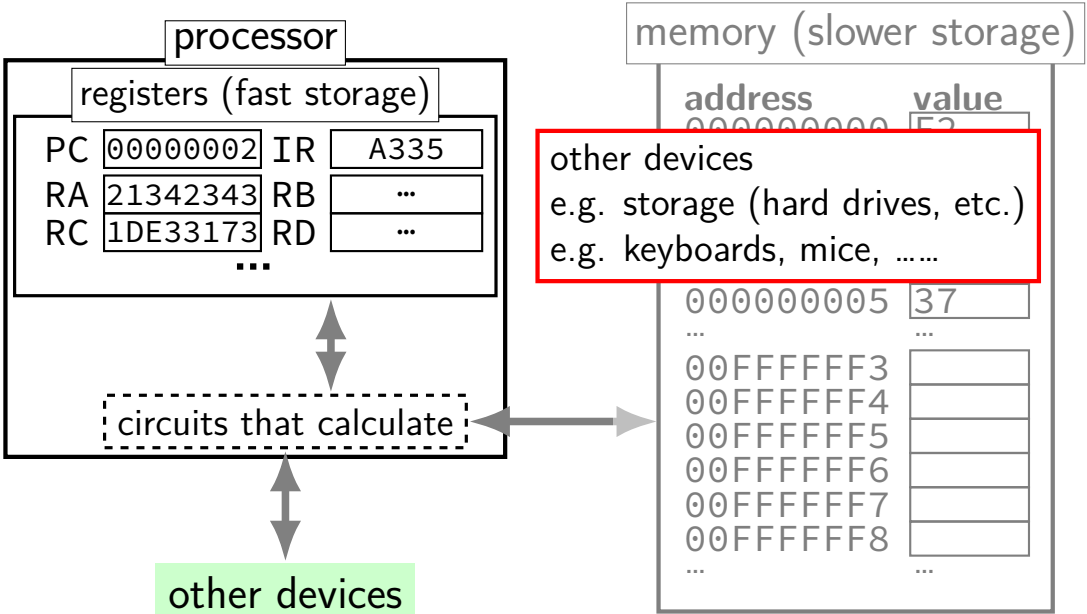
our machine model



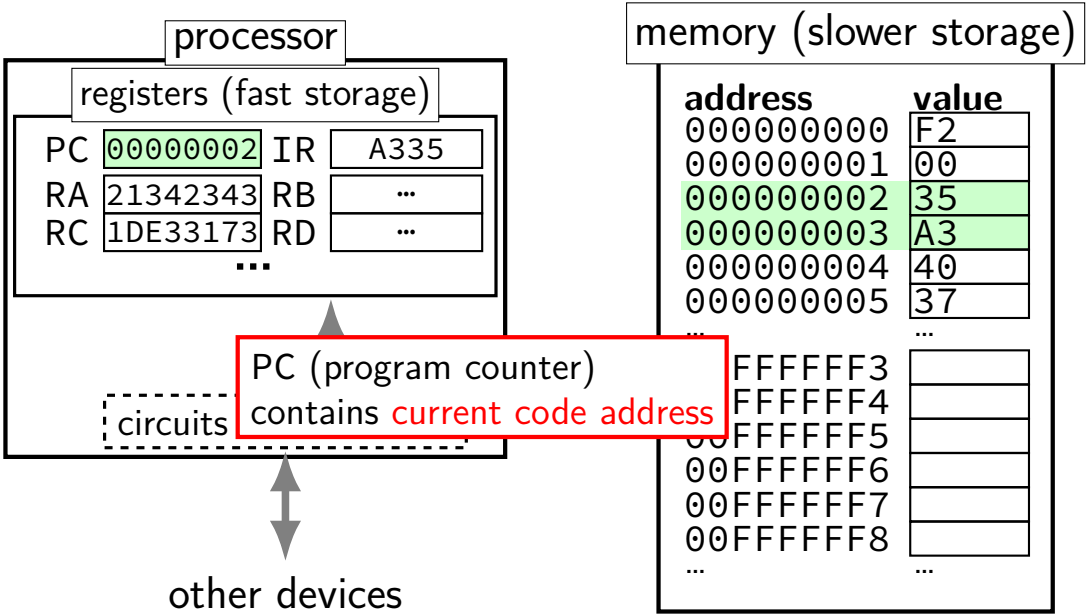
our machine model



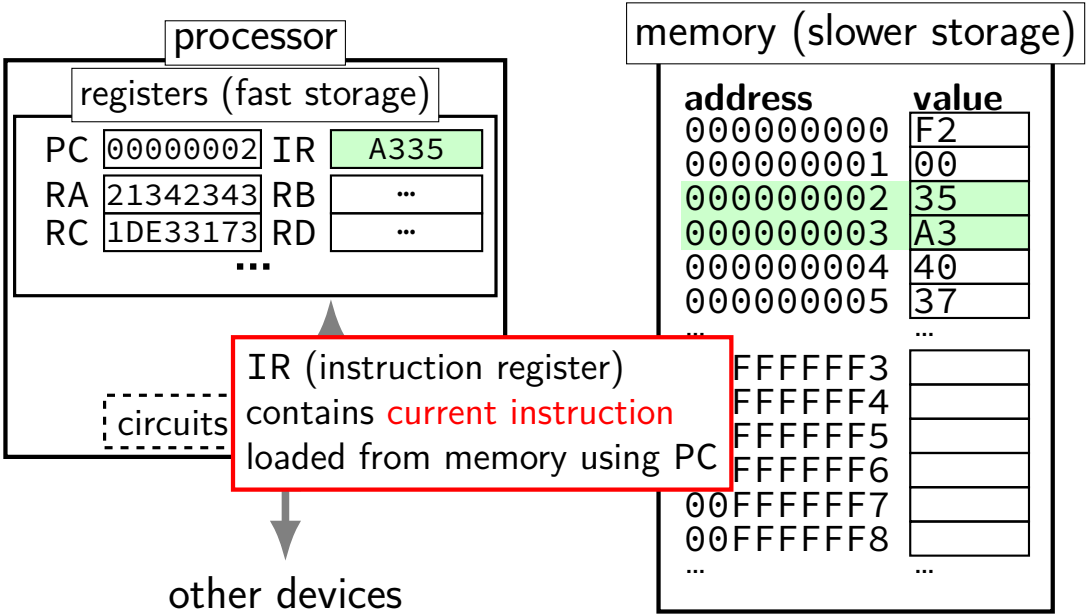
our machine model



our machine model



our machine model



fetch execute cycle

```
while (true) {
    IR <- memory[PC]
    execute instruction IR
    if (instruction didn't change PC)
        PC <- PC + length of instruction in IR
}
```

PC = program counter

IR = instruction register

instructions — one operation

- in machine code: represented by bits
- in assembly language: reprsented by text

example instructions

(in assembly language)

x86 example:

```
add ecx, ebx
add ecx, 1
```

(`ecx` and `ebx` are registers)

IBCM example:

```
load 100
add 200
store 300
```

(implicitly uses special "accumulator" register)

IBCM simulators

toy assembly language **IBCM**

no physical implementation, so...

simulators (all point to same implementation):

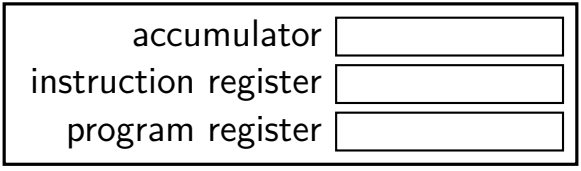
```
https://www.cs.virginia.edu/~cs216/ibcm/
https://people.virginia.edu/~asb2t/ibcm/
```

works in browser

will do bad things if your program doesn't terminate

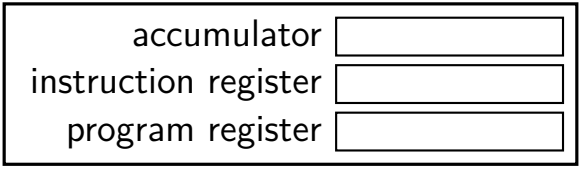
(turn off the simulated machine)

IBCM machine state

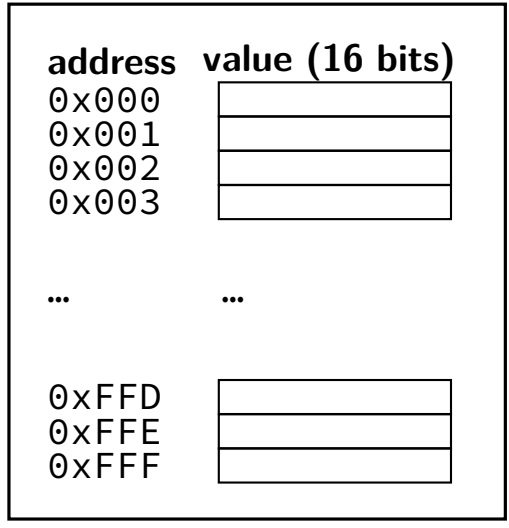


3 registers (16 bits each)

IBCM machine state



3 registers (16 bits each)



memory

4096 (2^{12}) 16-bit words

on words

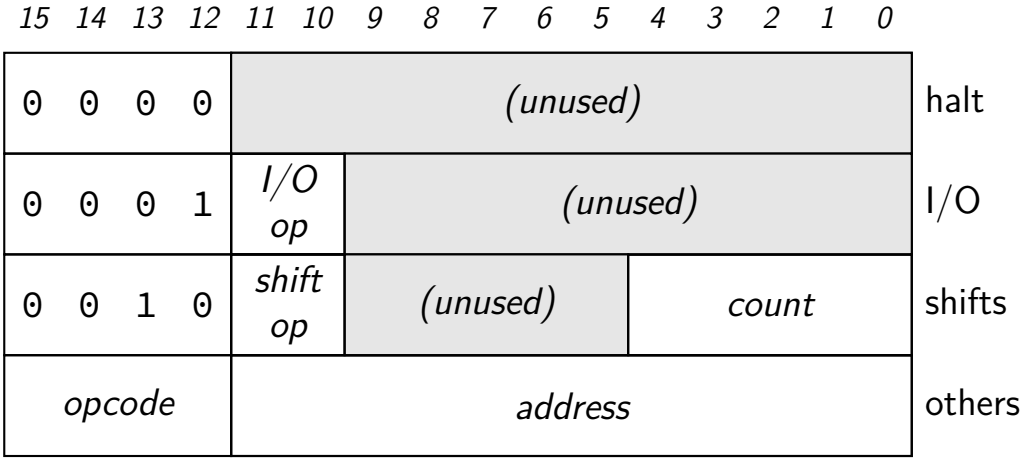
we deal with a lot of 16-bit values

"natural" size of this machine

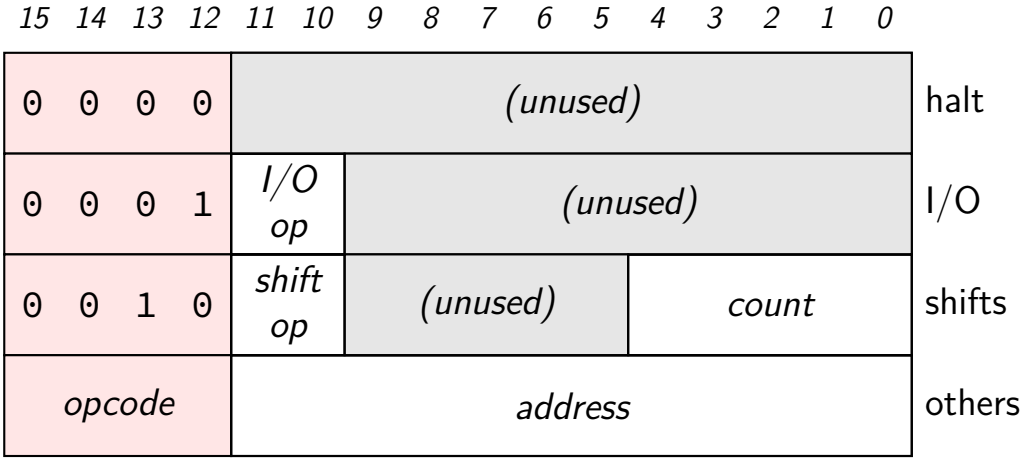
- size of registers
- size of memory accesses
- ...

convention: natural size called word

IBCM instruction format



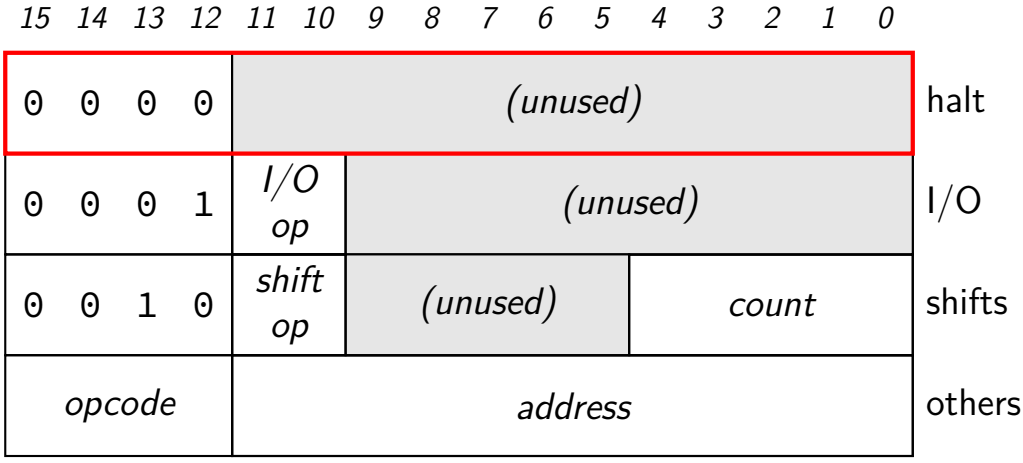
IBCM instruction format



opcode

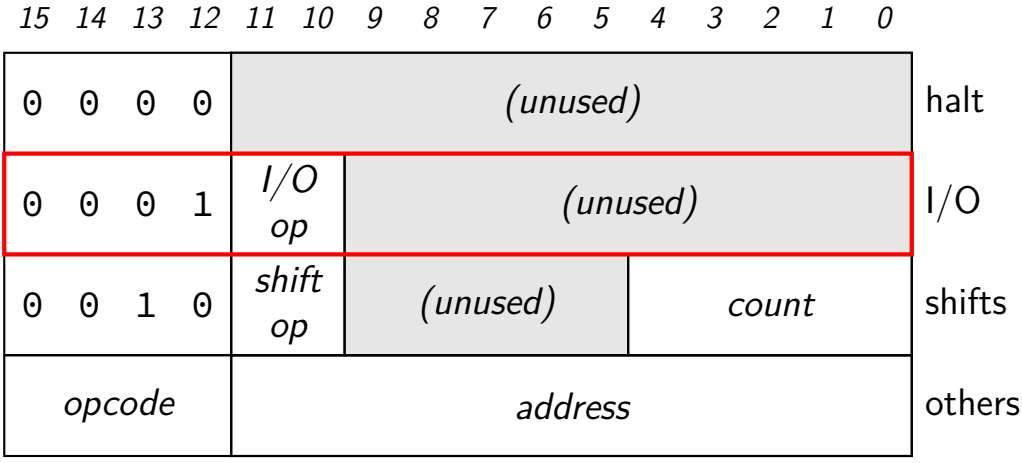
which instruction?

IBCM instruction format

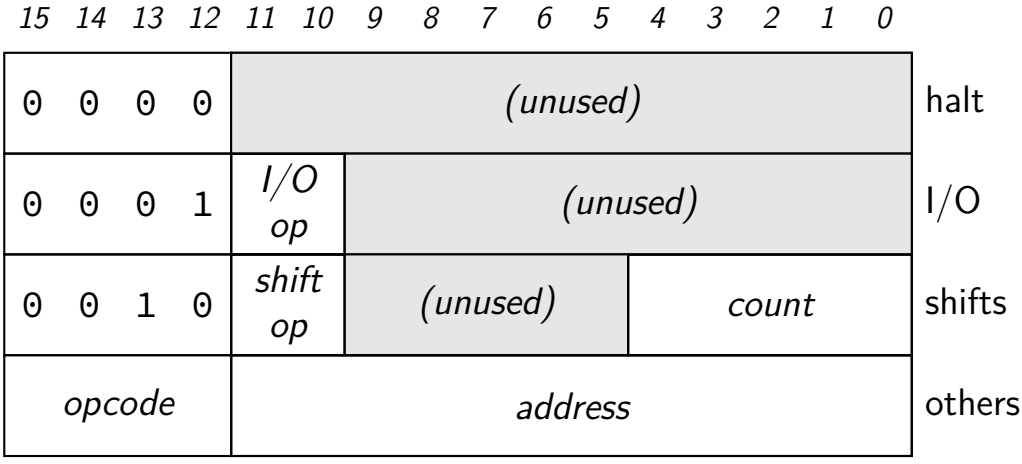


`halt` — opcode `0`
stops the machine

IBCM instruction format

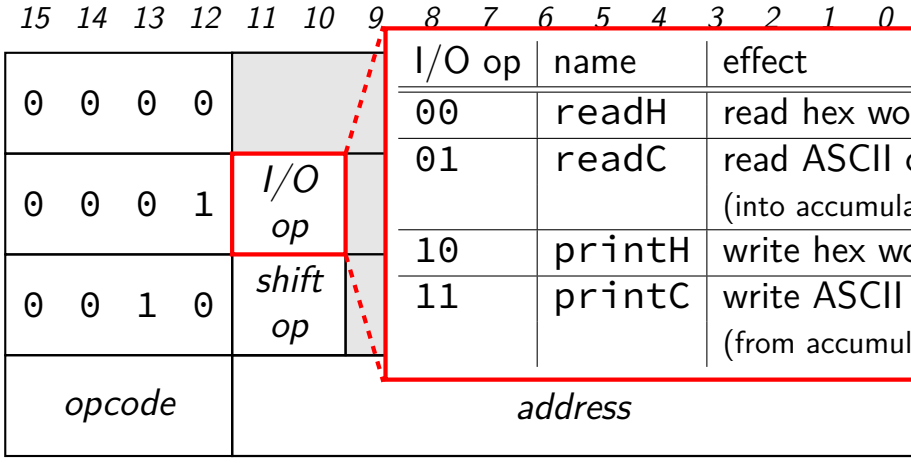


IBCM instruction format



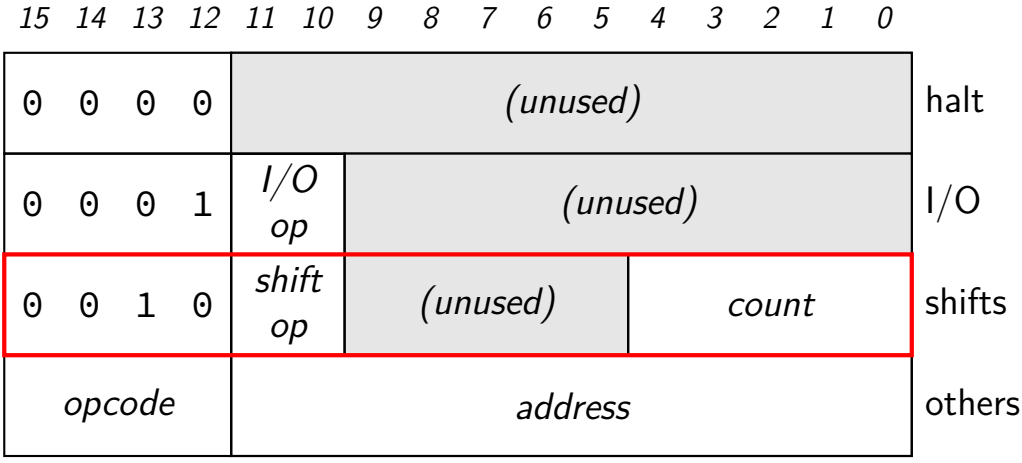
I/O operation – opcode 1
4 types ("I/O op" bits)
into or out of *accumulator*

IBCM instruction format



I/O op	name	effect
00	readH	read hex word
01	readC	read ASCII character (into accumulator bits 0–7)
10	printH	write hex word
11	printC	write ASCII charcater (from accumulator bits 0–7)

IBCM instruction format



`shift` — opcode 2

4 types ("shift op") move bits of accumulator around

count is number of places to move

shifts

has *shift op* (2 bits) and *count* (3 bits)

example: accumulator=`0000 1111 0000 1111`; *count*=3

shift op	name	example result	
`00`	shift left	shiftL	`0111 1000 0111 1000`
`01`	shift right	shiftR	`0000 0001 1110 0001`
`10`	rotate left	rotL	`0111 1000 0111 1000`
`11`	rotate right	rotR	`1110 0001 1110 0001`

shift: move bits, fill with `0`s

rotate: move bits, wrap around

other instructions

use accumulator (a or "acc") and/or address *in instruction*

op	*name*	*pseudocode*	*description*
3	`load`	`a ← mem[addr]`	load acc from memory
4	`store`	`a ← mem[addr]`	store acc to memory
5	`add`	`a ← a + mem[addr]`	add memory to acc
6	`sub`	`a ← a + mem[addr]`	subtract mememory from acc
7	`and`	`a ← a ∧ mem[addr]`	logical 'and' memory into acc
8	`or`	`a ← a ∨ mem[addr]`	logical 'or' memory into acc
9	`xor`	`a ← a ⊕ mem[addr]`	logical 'xor' memory into acc
A	`not`	`a ← ~a`	logical complement acc
B	`nop`	—	do nothing ('no operation')
C	`jmp`	`PC ← addr`	jump to `addr`
D	`jmpe`	`if a == 0: PC ← addr`	jump to `addr` if acc is 0
E	`jmpl`	`if a < 0: PC ← addr`	jump to `addr` if acc is negative
F	`brl`	`a ← PC + 1; PC ← addr`	jump to `addr` and set acc to the address following the `brl`

brl

"**br**anch and **l**ink"

`a` $\leftarrow$ `PC + 1; PC` $\leftarrow$ `addr`

used to implement method calls:

example: `addr` is the address of a method

`a` becomes the `return address`

instruction to execute *after the method returns*
issue in IBCM: jumping to `a`???

ICBM assembly language

don't have an assembler implemented

...but let's see what an assembly language would look like

ICBM assembler

assembly: `load 0x100`
→ opcode=3, addr=`0x100`
machine code: `0011 000100000000`

assembly: `add 0x200`
→ opcode=5, addr=`200`
machine code: `0101 001000000000`

assembly: `jmpe 0x442`
→ opcode=D, addr=`442`
machine code: `1101 010001000010`

ICBM assembler

assembly: `load 0x100`
→ opcode=3, addr=0x100
machine code: `0011 000100000000`

assembly: `add 0x200`
→ opcode=5, addr=200
machine code: `0101 001000000000`

assembly: `jmpe 0x442`
→ opcode=D, addr=442
machine code: `1101 010001000010`

work with hard-coded addresses?

how to set initial values?

labels: addresses as names

```
add     100     // addr 0: a += mem[100]
jmpl    3       // addr 1: if a < 0: goto 3
jmp     0       // addr 2: [otherwise] goto 0
nop             // addr 3: do nothing
```

labels: addresses as names

```
add     100     // addr 0: a += mem[100]
jmpl    3       // addr 1: if a < 0: goto 3
jmp     0       // addr 2: [otherwise] goto 0
nop             // addr 3: do nothing
```

```
start   add     100     // addr 0: a += mem[100]
        jmpl    end     // addr 1: if a < 0: goto 3
        jmp     start   // addr 2: [otherwise] goto 0
end     nop             // addr 3: do nothing
```

labels: addresses as name

```
start   add     100     // addr 0: a += mem[100]
        jmpl    end     // addr 1: if a < 0: goto 3
        jmp     start   // addr 2: [otherwise] goto 0
end     nop             // addr 3: do nothing
```

name for a memory address

address of instruction or of data

replaced by address when executable is produced

ICBM assembler

assembly: `load 0x100`
→ opcode=3, addr=0x100
machine code: `0011 000100000000`

assembly: `add 0x200`
→ opcode=5, addr=200
machine code: `0101 001000000000`

assembly: `jmpe 0x442`
→ opcode=D, addr=442
machine code: `1101 010001000010`

work with hard-coded addresses?

how to set initial values?

assembly directives

not everything in assembly is instructions

program data, strings, etc.

assemblers have directives

processed by assembler to produce special output

assembly directives

not everything in assembly is instructions

program data, strings, etc.

assemblers have directives

processed by assembler to produce special output

dw directive ("define word")

assembly directives

not everything in assembly is instructions

program data, strings, etc.

assemblers have directives

processed by assembler to produce special output

dw directive ("define word")

```
i dw 75
```

place the value 75 in memory
name the address where it is placed `i`

example with dw

```
loop    load hundred    // a ← 100
        jmpl end        // if a < 0: goto end
        printH          // print a
        sub one         // a ← a - 1
        jmp loop
end     halt

hundred dw 100
one     dw 1
```

```
int a = 100;
while (a >= 0) {
    print a;
    a -= 1;
}
```

variables with dw

```
        load i
        add j
        store i

        load j
        sub i
        sub i
        store j

i       dw 10
j       dw 20
```

```
int i = 10, j = 20;
i += j;
j -= i;
j -= i;
```

IBCM decoding

value
0000
000f
0005
3041
5002
1800
2403
0000

IBCM decoding

value
0000
000f
0005
3041
5002
1800
2403
0000

most significant 4 bits = opcode

0 — halt

1 — some kind of I/O

3 — load

5 — add

IBCM decoding

value	as instruction
`0000`	`halt`
`000f`	`halt`
`0005`	`halt`
`3041`	`load ?`
`5002`	`add ?`
`1800`	`?? I/O`
`2403`	`?? shift`
`0000`	`halt`

most significant 4 bits = opcode

`0` — halt

1 — some kind of I/O

3 — load

5 — add

IBCM decoding

value	as instruction
0000	halt
000f	halt
0005	halt
3041	load ?
5002	add ?
1800	?? I/O
2403	?? shift
0000	halt

halt — rest of instruction ignored

IBCM decoding

value	as instruction
0000	halt
000f	halt
0005	halt
3041	load 0x41
5002	add 0x2
1800	?? I/O
2403	?? shift
0000	halt

load/add — rest is address

IBCM decoding

value	as instruction
0000	halt
000f	halt
0005	halt
3041	load 0x41
5002	add 0x2
1800	printH
2403	shiftR ?
0000	halt

I/O: bits 10–11 = 10 → printH
shift: bits 10–11 = 01 → shiftR

IBCM decoding

value	as instruction
0000	halt
000f	halt
0005	halt
3041	load 0x41
5002	add 0x2
1800	printH
2403	shiftR 3
0000	halt

shift amount in bottom 4 bits

IBCM sample program

addr.	value
000	3000
001	5000
002	6001
003	8003
004	a000
005	4000
006	f000

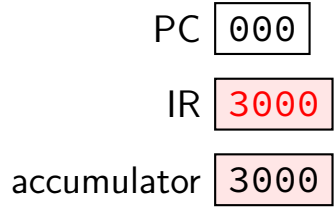
PC 000

IR ????

accumulator 3000

IBCM sample program

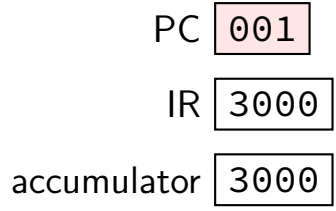
addr.	value	as instruction
000	3000	load 0
001	5000	add 0
002	6001	sub 1
003	8003	or 3
004	a000	not
005	4000	store 0
006	f000	brl 0



accumulator $\leftarrow$ 0x3000 $=$ memory[0]

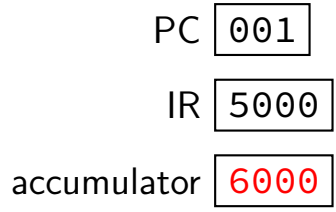
IBCM sample program

addr.	value	as instruction
000	3000	load 0
001	5000	add 0
002	6001	sub 1
003	8003	or 3
004	a000	not
005	4000	store 0
006	f000	brl 0



IBCM sample program

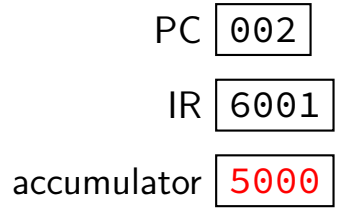
addr.	value	as instruction
000	3000	load 0
001	5000	add 0
002	6001	sub 1
003	8003	or 3
004	a000	not
005	4000	store 0
006	f000	brl 0



accumulator $\leftarrow$ 0x6000 $=$ 0x3000 $+$ memory[0]

IBCM sample program

addr.	value	as instruction
000	3000	load 0
001	5000	add 0
002	6001	sub 1
003	8003	or 3
004	a000	not
005	4000	store 0
006	f000	brl 0



accumulator $\leftarrow$ 0x1000 $=$ 0x6000 - memory[1]

IBCM sample program

addr.	value	as instruction
000	3000	load 0
001	5000	add 0
002	6001	sub 1
003	8003	or 3
004	a000	not
005	4000	store 0
006	f000	brl 0

PC 003

IR 8003

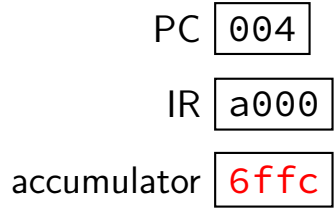
accumulator 9003

accumulator $\leftarrow$ 0x9003 = 0x1000 OR memory[3]

"or" — bitwise or:
bit x set in result if set in either operand

IBCM sample program

addr.	value	as instruction
000	3000	load 0
001	5000	add 0
002	6001	sub 1
003	8003	or 3
004	a000	not
005	4000	store 0
006	f000	brl 0



accumulator $\leftarrow$ 0x6ffc = NOT 0x9003

"not" — flip every bit

IBCM sample program

addr.	value	as instruction
000	6ffc	load 0sub FFC
001	5000	add 0
002	6001	sub 1
003	8003	or 3
004	a000	not
005	4000	store 0
006	f000	brl 0

PC 005

IR 4000

accumulator 6ffc

memory[0] $\leftarrow$ accumulator

IBCM sample program

addr.	value	as instruction
000	6ffc	load 0sub FFC
001	5000	add 0
002	6001	sub 1
003	8003	or 3
004	a000	not
005	4000	store 0
006	f000	brl 0

PC 006

IR f000

accumulator 0007

accumulator $\leftarrow$ PC + 1PC $\leftarrow$ 0

IBCM sample program

addr.	value	as instruction
000	6ffc	load 0sub FFC
001	5000	add 0
002	6001	sub 1
003	8003	or 3
004	a000	not
005	4000	store 0
006	f000	brl 0

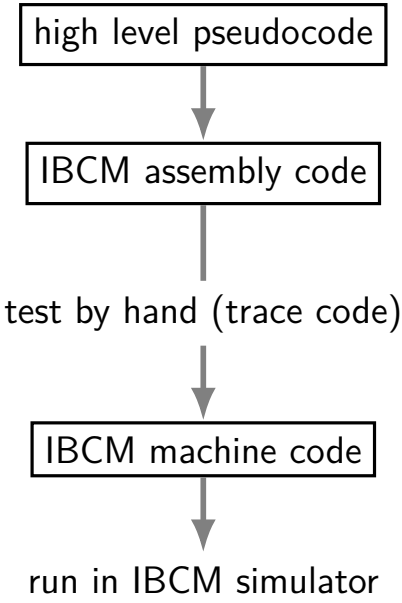
PC 001

IR 6ffc

accumulator ????

accumulator $\leftarrow$??? $=$ 0x0007 - memory[0xFFC]

writing IBCM




```
i = 1;
while (i < max) ...
```

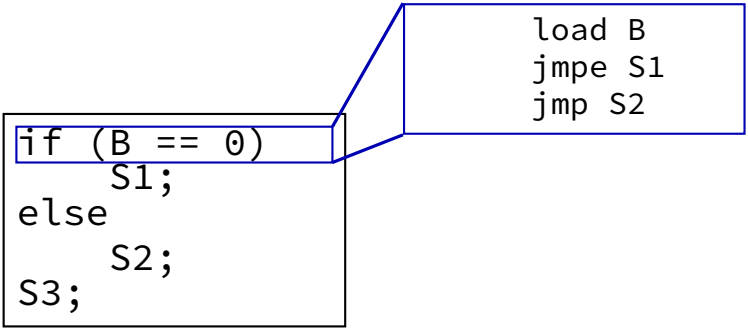
```
load  one
store i
...
```

```
3016
4005
...
```

example: if/else

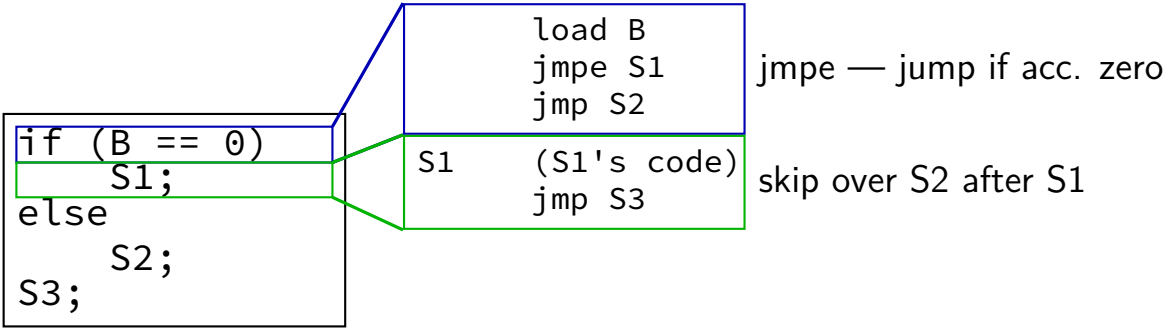
```
if (B == 0)
    S1;
else
    S2;
S3;
```

example: if/else

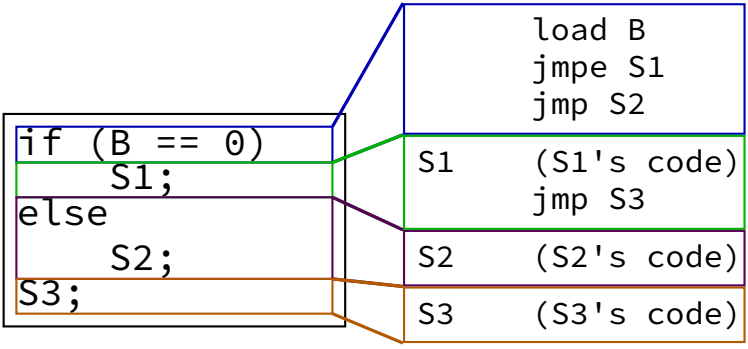


jmpe — jump if acc. zero

example: if/else



example: if/else



jmpe — jump if acc. zero

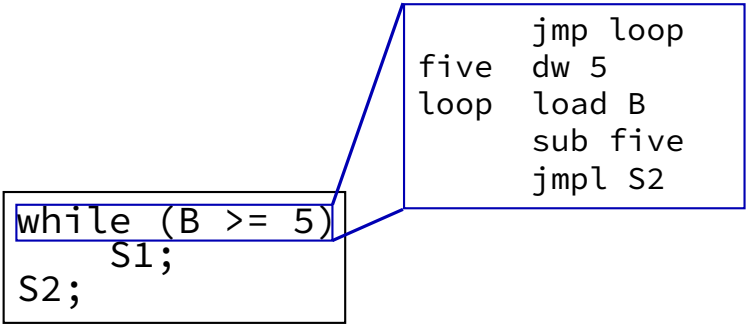
skip over S2 after S1

can omit jump to S3, since it's right after

example: while

```
while (B >= 5)
    S1;
S2;
```

example: while

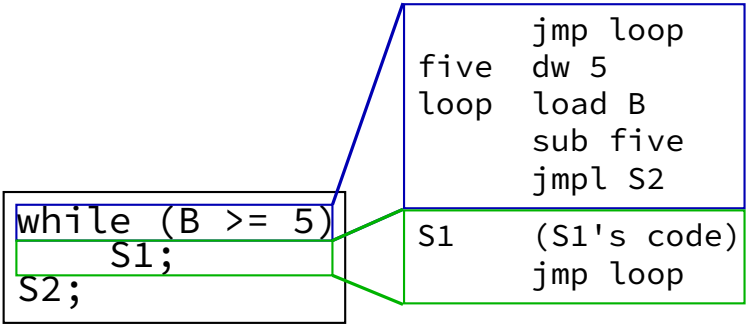


need constant '5'

B - 5 $<$ 0 $\rightarrow$

done with loop

example: while

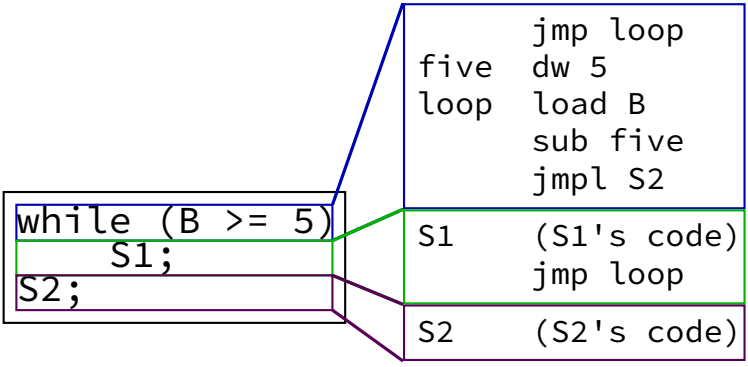


need constant '5'

B - 5 $<$ 0 $\rightarrow$

done with loop

example: while



need constant '5'

B - 5 $<$ 0 $\rightarrow$

done with loop

example: sum

the task:

read in integer n from keyboard

compute sum of integers 1 to n (inclusive)

print sum

halt

sum psuedocode

```
read n;
i = 1;          // index in the array
s = 0;          // ongoing sum
while (i <= n) {
  s += i;
  i += 1;
}
print s;
```

translating sum (1)

```
read n;
i = 1;
s = 0;
while (i <= n) {
  s += i;
  i += 1;
}
print s;
```

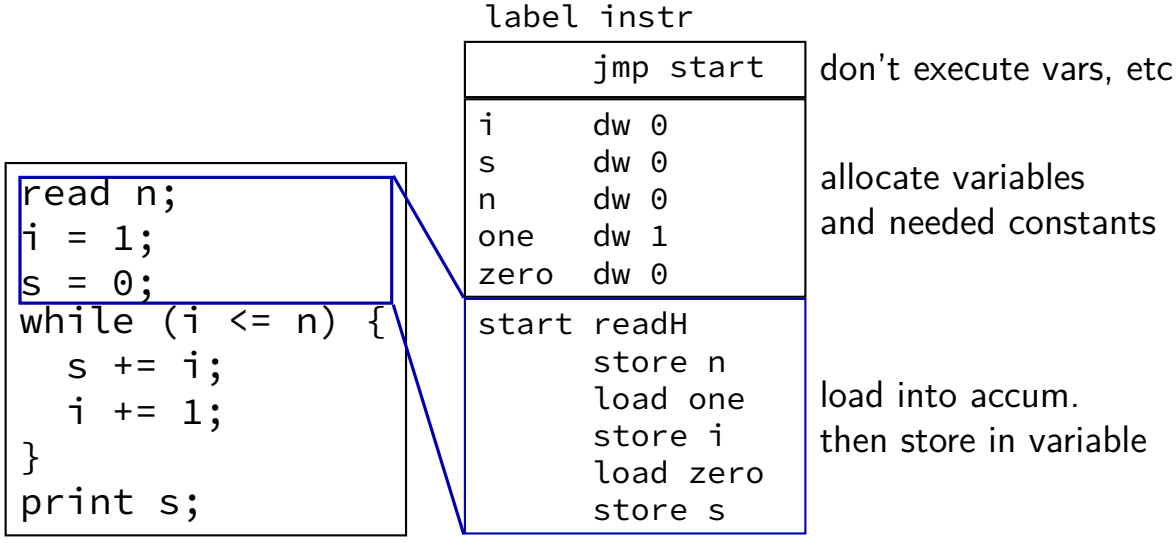
translating sum (1)

```
read n;
i = 1;
s = 0;
while (i <= n) {
  s += i;
  i += 1;
}
print s;
```

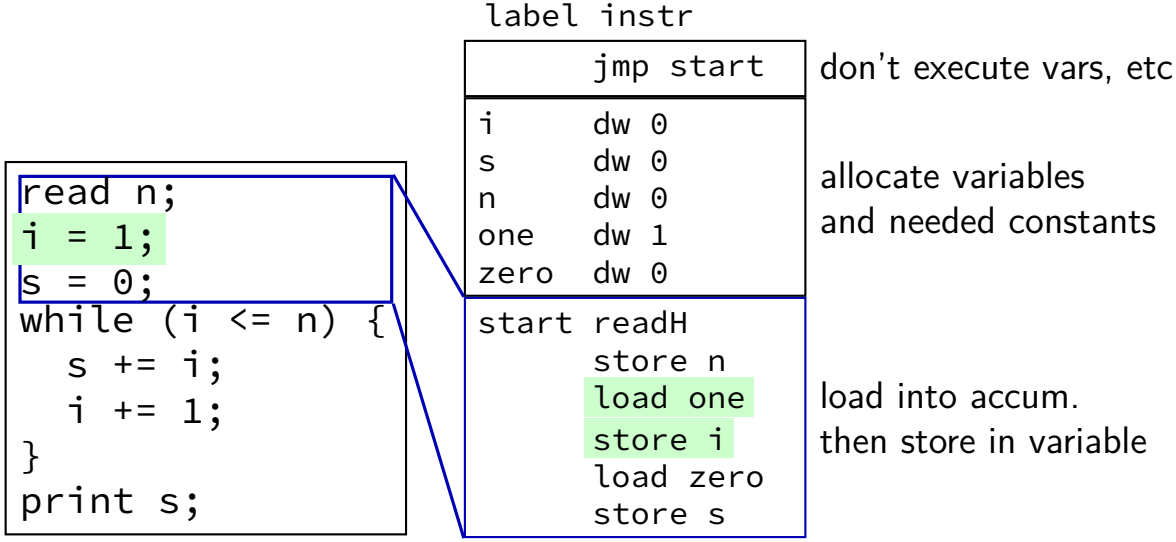
```
label instr
i       dw 0
s       dw 0
n       dw 0
one     dw 1
zero    dw 0
```

allocate variables
and needed constants

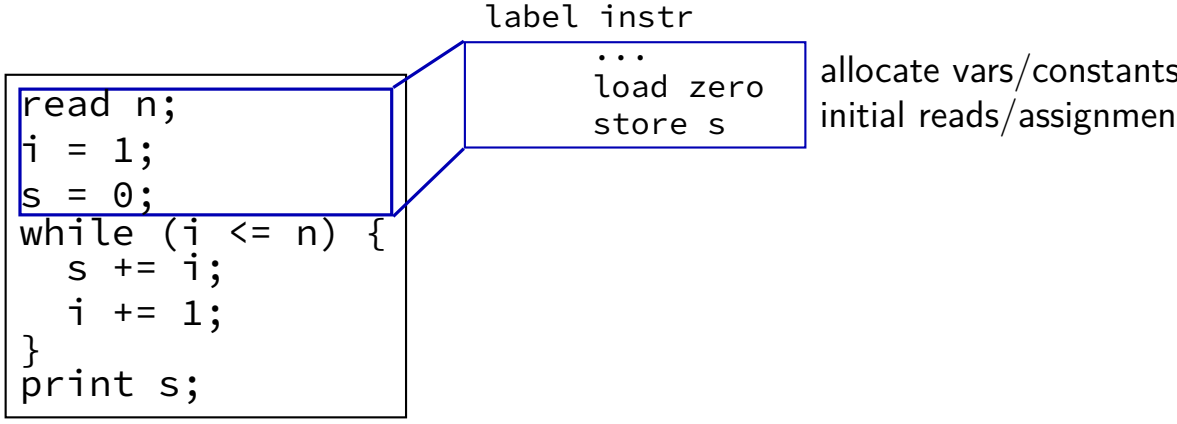
translating sum (1)



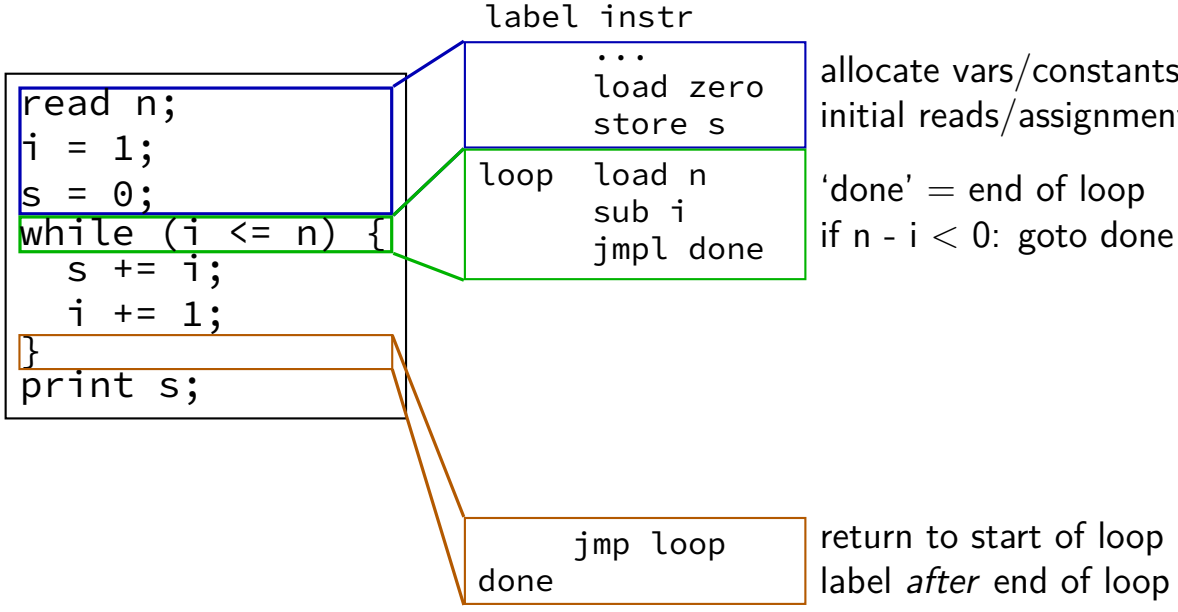
translating sum (1)



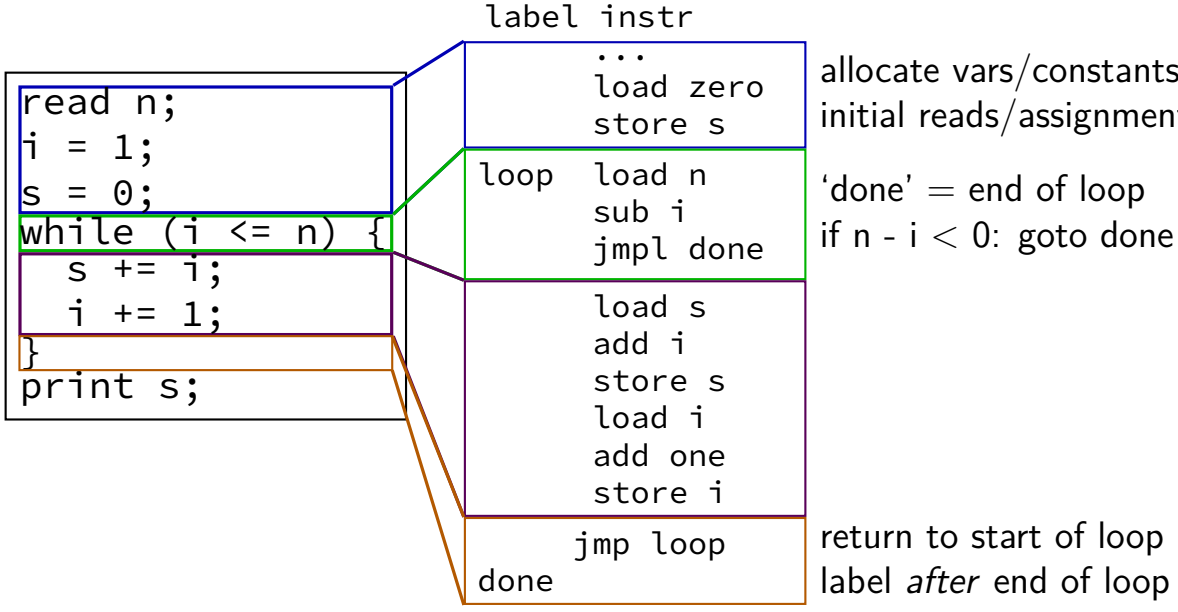
translating sum (2)



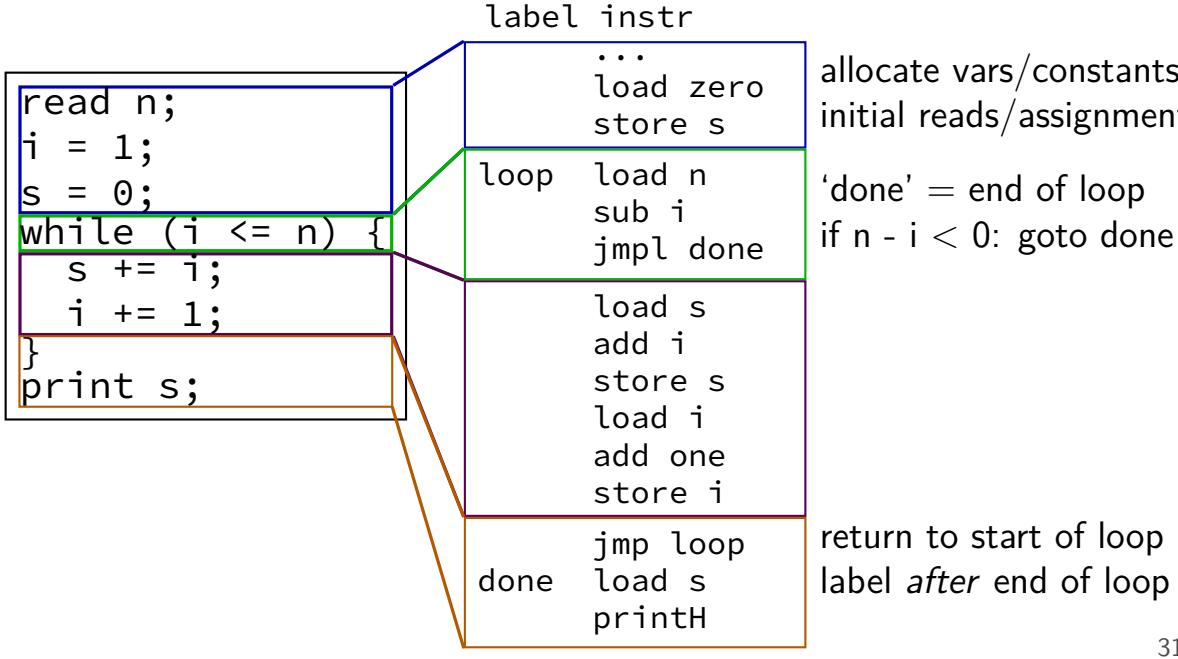
translating sum (2)



translating sum (2)



translating sum (2)



code is just data

IBCM had array of 'words' (16-bit values)

could be data or code or both

how to know which?

- what is the machine trying to do when it read/writes it?
  (e.g. `jmp` or `load`)

how typical modern computers work

code+data together machine: 'von Neumann architecture'

- seperate code+data memory: 'Harvard architecture'

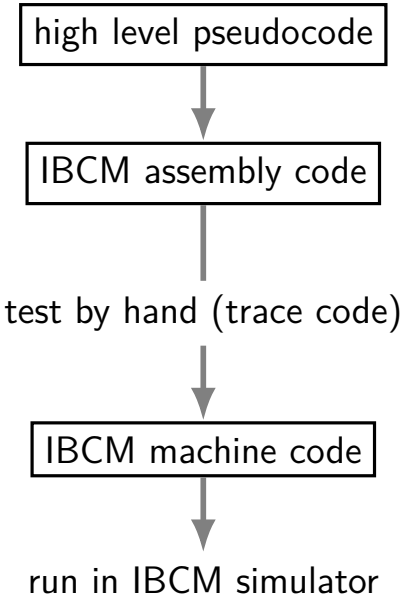
IBCM can do...

IBCM can do “anything”

formally: Turing complete (if extended to infinitely large memory)

- formal definition: see CS 3102

writing IBCM




```
i = 1;
while (i < max) ...
```

```
load  one
store i
...
```

```
3016
4005
...
```

IBCM tips

write assembly code first

- use comments (for you and us)

write machine code last

check functionality in simulator

NB: simulator does not accept blank/comment lines

simulators and infinite loops

online simulator won't like infinite loops

- likely reason for web page just not responding

debugging advice

check program logic

- correct conditions for jmpl/jmpe?

check machine code translation

- follow decoding steps
- verify addresses

mssing from IBCM

multiply, divide

floating point

bigger addresses or values

more registers (and ability to specify registers)

...

implementing IBCM

```
unsigned short memory[4096];
unsigned short pc, ir, accum;
bool done = false;
while (!done) {
    ir = memory[pc];
    switch (extractOpcode(ir)) {
    case 0:
        // halt
        done = true;
        break;
    case 1:
        // I/O
        ...
```

implementing IBCM

```
unsigned short memory[4096];
unsigned short pc, ir, accum;
bool done = false;
while (!done) {
    ir = memory[pc];
    switch (extractOpcode(ir)) {
    case 0:
        // halt
        done = true;
        break;
    case 1:
        // I/O
        ...
```

extracting parts of instructions

assuming instruction in `instr`:

```
unsigned int opcode = (instr >> 12) & 0x000f;
unsigned int ioOrShiftOp = (instr >> 10) & 0x0003;
unsigned int address = instr & 0x0fff;
unsigned int shiftCount = instr & 0x000f;
```

`>>` — shift right

`&` — bitwise (bit-by-bit) and

extracting parts of instructions

assuming instruction in `instr`:

```
unsigned int opcode = (instr >> 12) & 0x000f;
unsigned int ioOrShiftOp = (instr >> 10) & 0x0003;
unsigned int address = instr & 0x0fff;
unsigned int shiftCount = instr & 0x000f;
```

`>>` — shift right

`&` — bitwise (bit-by-bit) and

---

but, isn't this very cumbersome???

encoding instructions

assuming instruction in `instr`:

```
unsigned int instr = (opcode << 12) | address;
unsigned int instr = (opcode << 12) | (ioOrShiftOp <
```

`<<` — shift right

`|` — bitwise (bit-by-bit) or

encoding instructions

assuming instruction in `instr`:

```
unsigned int instr = (opcode << 12) | address;
unsigned int instr = (opcode << 12) | (ioOrShiftOp <
```

<< — shift right

| — bitwise (bit-by-bit) or

---

but, isn't this very cumbersome???

C++ support for bit-extraction (1)

```
// assumes unsigned short is 16 bits
// and most common compiler convention for ordering
union ibcm_instruction {
    unsigned short value;
    struct { unsinged op: 4, ioOp: 2,
                     unused: 10; } io;
    struct { unsinged op: 4, shiftOp: 2,
                     shiftCount: 5; } shifts;
    struct { unsigned op: 4,
                     address: 12; } others;
};
```

C++ support for bit-extraction (2)

```
union ibcm_instruction i;
i.value = memory[pc];
switch (i.others.op) {
    ...
}
```

on bit fields

`value : 4` — called ‘a bit field’

technically, order of bits can vary between compilers