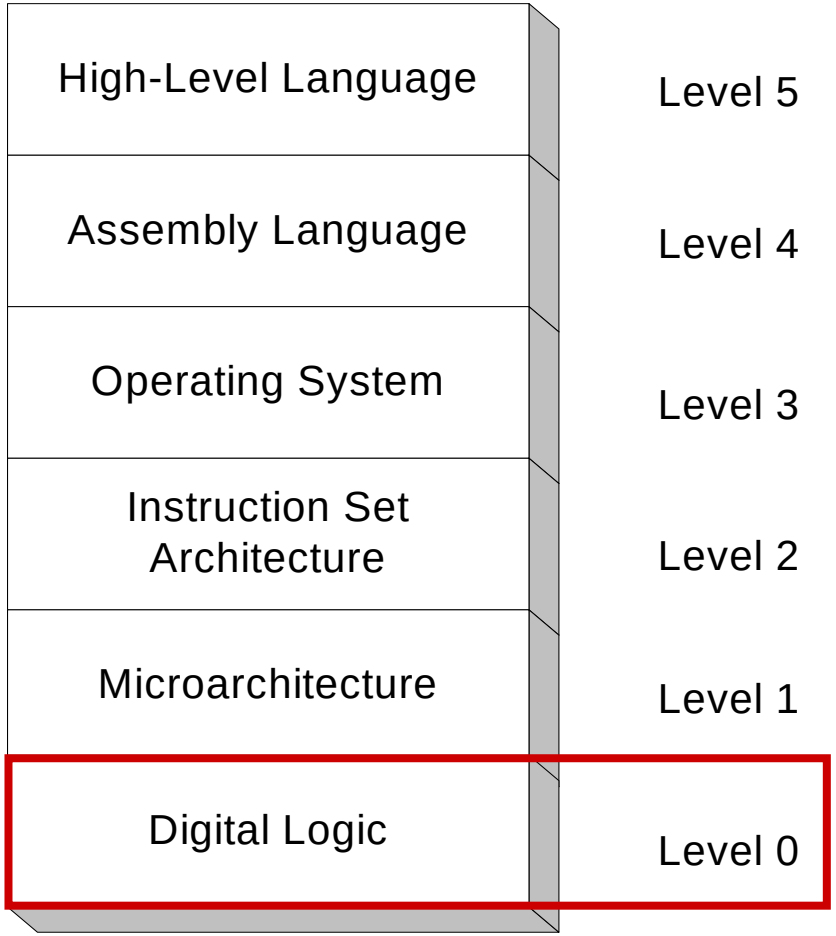
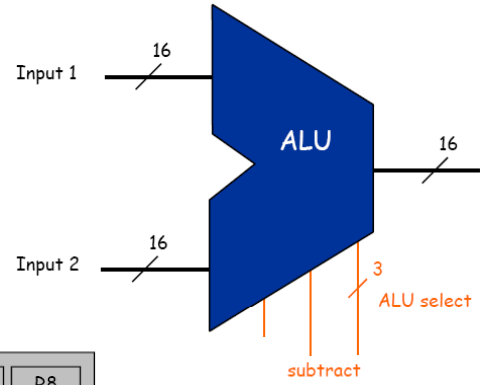
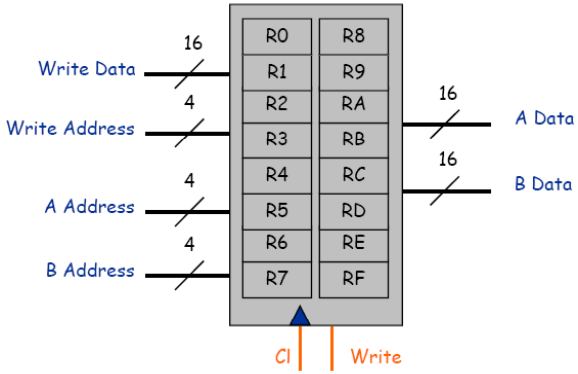
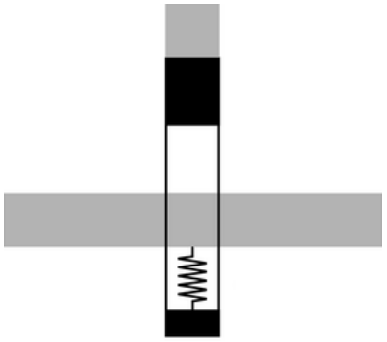
Virtual machines

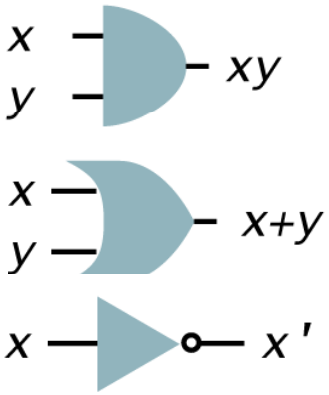
Abstractions for computers







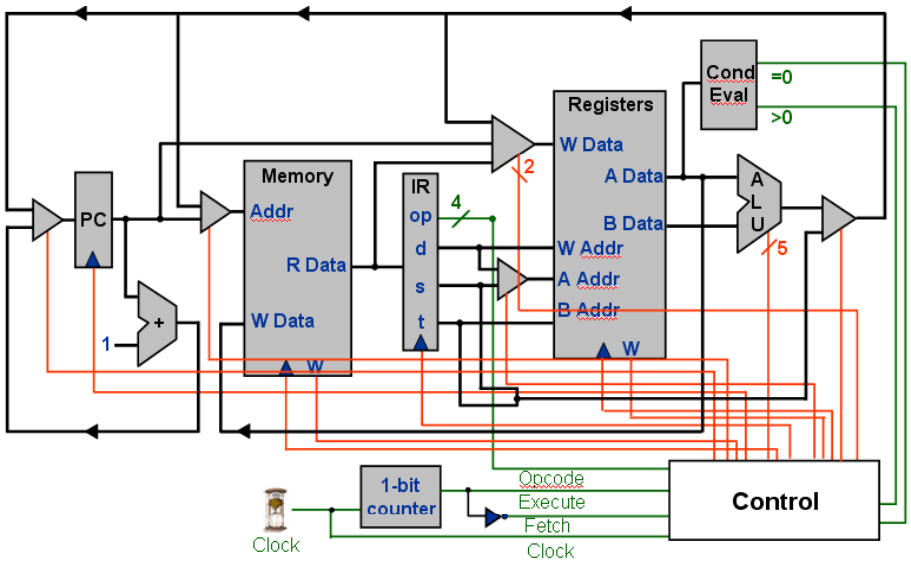




Virtual machines

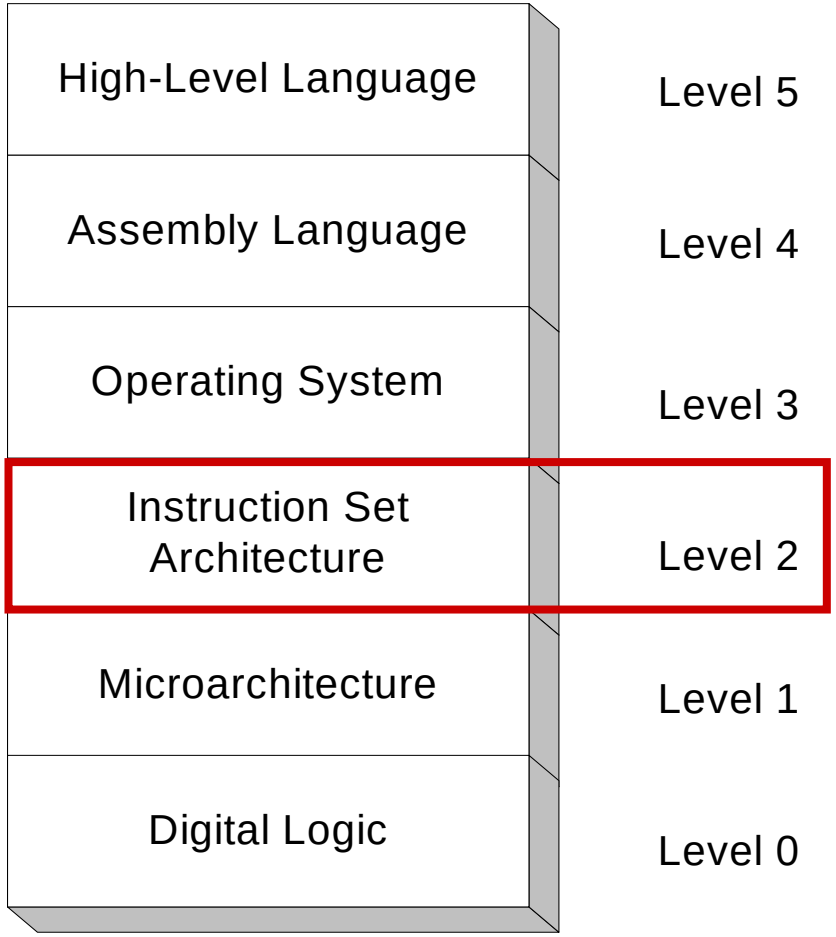
Abstractions for computers

High-Level Language	Level 5
Assembly Language	Level 4
Operating System	Level 3
Instruction Set Architecture	Level 2
Microarchitecture	Level 1
Digital Logic	Level 0



Virtual machines

Abstractions for computers



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Format 1	opcode				dest d				source s				source t			
Format 2	opcode				dest d				addr							

#	Operation	Emt	Pseudocode
0:	halt	1	exit(0)
1:	add	1	R[d] ← R[s] + R[t]
2:	subtract	1	R[d] ← R[s] - R[t]
3:	and	1	R[d] ← R[s] & R[t]
4:	xor	1	R[d] ← R[s] ^ R[t]
5:	shift left	1	R[d] ← R[s] << R[t]
6:	shift right	1	R[d] ← R[s] >> R[t]
7:	load addr	2	R[d] ← addr
8:	load	2	R[d] ← mem[addr]
9:	store	2	mem[addr] ← R[d]
A:	load indirect	1	R[d] ← mem[R[t]]
B:	store indirect	1	mem[R[t]] ← R[d]
C:	branch zero	2	if (R[d] == 0) pc ← addr
D:	branch positive	2	if (R[d] > 0) pc ← addr
E:	jump register	1	pc ← R[t]
F:	jump and link	2	R[d] ← pc; pc ← addr

10: C020

20: 7101
21: 7A00
22: 7C00

23: 8DFF
24: CD29
25: 12AC
26: BD02
27: 1CC1
28: C023

29: FF2B
2A: 0000

Problems with programming using machine code

- Difficult to remember instructions
- Difficult to remember variables
- Hard to calculate addresses/relocate variables or functions
- Need to handle instruction encoding (e.g. jr Rt)

Table B.1 ARM instruction decode table.

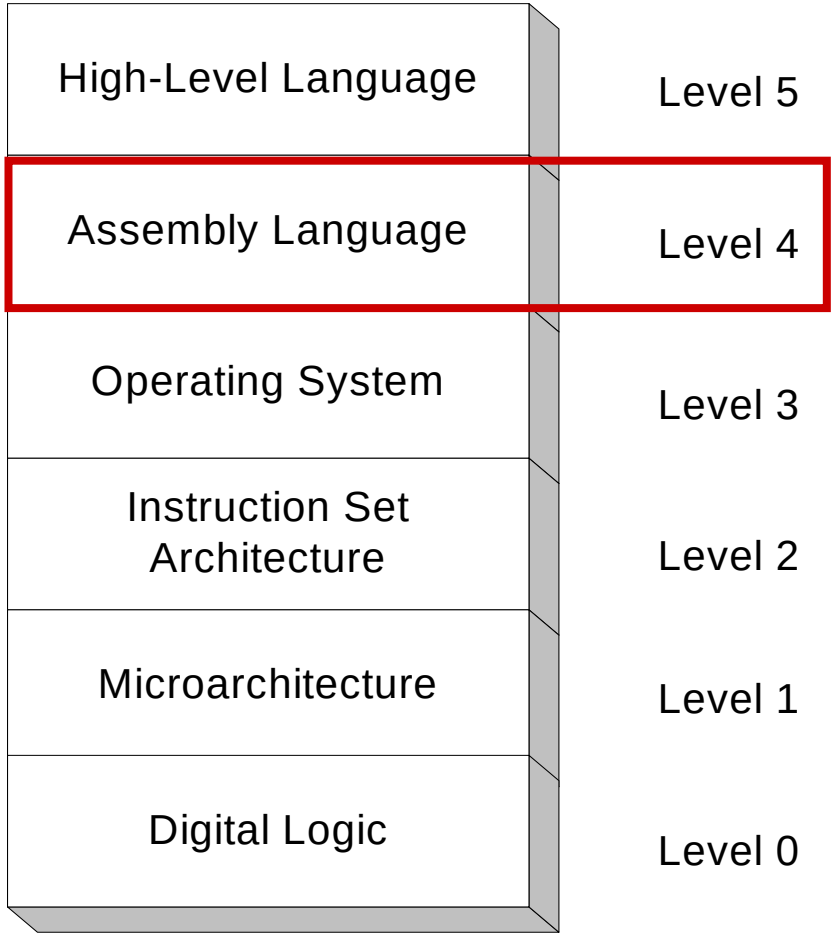
Instruction classes (indexed by *op*)	31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
AND | EOR | SUB | RSB | ADD | ADC | SBC | RSC	*cond*	0 0 0 0	*op*	*S*	*Rn*	*Rd*	*shift_size*	*shift*	0	*Rm*
AND | EOR | SUB | RSB | ADD | ADC | SBC | RSC	*cond*	0 0 0 0	*op*	*S*	*Rn*	*Rd*	*Rs* 0	*shift*	1	*Rm*
MUL	*cond*	0 0 0 0	0 0 0	*S*	*Rd*	0 0 0 0	*Rs* 1	0 0	1	*Rm*
MLA	*cond*	0 0 0 0	0 0 1	*S*	*Rd*	*Rn*	*Rs* 1	0 0	1	*Rm*
UMAAL	*cond*	0 0 0 0	0 1 0	0	*RdHi*	*RdLo*	*Rs* 1	0 0	1	*Rm*
UMULL | UMLAL | SMULL | SMLAL	*cond*	0 0 0 0	1 *op*	*S*	*RdHi*	*RdLo*	*Rs* 1	0 0	1	*Rm*
STRH | LDRH *post*	*cond*	0 0 0 0	*U* 0 0	*op*	*Rn*	*Rd*	0 0 0 0 1	0 1	1	*Rm*
STRH | LDRH *post*	*cond*	0 0 0 0	*U* 1 0	*op*	*Rn*	*Rd*	*immed* [7:4] 1	0 1	1	*immed* [3:0]
LDRD | STRD | LDRSB | LDRSH *post*	*cond*	0 0 0 0	*U* 0 0	*op*	*Rn*	*Rd*	0 0 0 0 1	1 *op*	1	*Rm*
LDRD | STRD | LDRSB | LDRSH *post*	*cond*	0 0 0 0	*U* 1 0	*op*	*Rn*	*Rd*	*immed* [7:4] 1	1 *op*	1	*immed* [3:0]
MRS Rd, cpsr | MRS Rd, spsr	*cond*	0 0 0 1	0 *op* 0	0	1 1 1 1	*Rd*	0 0 0 0	0 0 0	0	0 0 0 0
MSR cpsr, Rm | MSR spsr, Rm	*cond*	0 0 0 1	0 *op* 1	0	*f s x c*	1 1 1 1	0 0 0 0	0 0 0	0	*Rm*
BXJ	*cond*	0 0 0 1	0 0 1	0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 1	0	*Rm*
SMLAxy	*cond*	0 0 0 1	0 0 0	0	*Rd*	*Rn*	*Rs* 1	*y x*	0	*Rm*
SMLAWy	*cond*	0 0 0 1	0 0 1	0	*Rd*	*Rn*	*Rs* 1	*y* 0	0	*Rm*
SMULWy	*cond*	0 0 0 1	0 0 1	0	*Rd*	0 0 0 0	*Rs* 1	*y* 1	0	*Rm*
SMLALxy	*cond*	0 0 0 1	0 1 0	0	*RdHi*	*RdLo*	*Rs* 1	*y x*	0	*Rm*
SMULxy	*cond*	0 0 0 1	0 1 1	0	*Rd*	0 0 0 0	*Rs* 1	*y x*	0	*Rm*
TST | TEQ | CMP | CMN	*cond*	0 0 0 1	0 *op*	1	*Rn*	0 0 0 0	*shift_size*	*shift*	0	*Rm*
ORR | BIC	*cond*	0 0 0 1	1 *op* 0	*S*	*Rn*	*Rd*	*shift_size*	*shift*	0	*Rm*
MOV | MVN	*cond*	0 0 0 1	1 *op* 1	*S*	0 0 0 0	*Rd*	*shift_size*	*shift*	0	*Rm*
BX | BLX	*cond*	0 0 0 1	0 0 1	0	1 1 1 1	1 1 1 1	1 1 1 1	0 *op*	1	*Rm*
CLZ	*cond*	0 0 0 1	0 1 1	0	1 1 1 1	*Rd*	1 1 1 1	0 0 0	1	*Rm*
QADD | QSUB | QDADD | QDSUB	*cond*	0 0 0 1	0 *op*	0	*Rn*	*Rd*	0 0 0 0	0 1 0	1	*Rm*
BKPT	1 1 1 0	0 0 0 1	0 0 1	0	*immed*[15:4]			0 1 1	1	*immed* [3:0]
TST | TEQ | CMP | CMN	*cond*	0 0 0 1	0 *op*	1	*Rn*	0 0 0 0	*Rs* 0	*shift*	1	*Rm*
ORR | BIC	*cond*	0 0 0 1	1 *op* 0	*S*	*Rn*	*Rd*	*Rs* 0	*shift*	1	*Rm*
MOV | MVN	*cond*	0 0 0 1	1 *op* 1	*S*	0 0 0 0	*Rd*	*Rs* 0	*shift*	1	*Rm*
SWP | SWPB	*cond*	0 0 0 1	0 *op* 0	0	*Rn*	*Rd*	0 0 0 0	1 0 0	1	*Rm*
STREX	*cond*	0 0 0 1	1 0 0	0	*Rn*	*Rd*	1 1 1 1	1 0 0	1	*Rm*
LDREX	*cond*	0 0 0 1	1 0 0	1	*Rn*	*Rd*	1 1 1 1	1 0 0	1	1 1 1 1

Table B.1 ARM instruction decode table. (*Continued.*)

Instruction classes (indexed by *op*)	31 30 29 28	27 26 25 24	23 22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
STRH | LDRH *pre*	*cond*	0 0 0 1	*U* 0	*W op*	*Rn*	*Rd*	0 0 0 0	1 0 1	1	*Rm*
STRH | LDRH *pre*	*cond*	0 0 0 1	*U* 1	*W op*	*Rn*	*Rd*	*immed* [7:4]	1 0 1	1	*immed* [3:0]
LDRD | STRD | LDRSB | LDRSH *pre*	*cond*	0 0 0 1	*U* 0	*W op*	*Rn*	*Rd*	0 0 0 0	1 1 *op*	1	*Rm*
LDRD | STRD | LDRSB | LDRSH *pre*	*cond*	0 0 0 1	*U* 1	*W op*	*Rn*	*Rd*	*immed* [7:4]	1 1 *op*	1	*immed* [3:0]
AND | EOR | SUB | RSB | ADD | ADC | SBC | RSC	*cond*	0 0 1 0	*op*	*S*	*Rn*	*Rd*	*rotate*	*immed*		
MSR cpsr, #imm | MSR spsr, #imm	*cond*	0 0 1 1	0 *op* 1	0	*f s x c*	1 1 1 1	*rotate*	*immed*		
TST | TEQ | CMP | CMN	*cond*	0 0 1 1	0 *op*	1	*Rn*	0 0 0 0	*rotate*	*immed*		
ORR | BIC	*cond*	0 0 1 1	1 *op* 0	*S*	*Rn*	*Rd*	*rotate*	*immed*		
MOV | MVN	*cond*	0 0 1 1	1 *op* 1	*S*	0 0 0 0	*Rd*	*rotate*	*immed*		
STR | LDR | STRB | LDRB *post*	*cond*	0 1 0 0	*U op T op*		*Rn*	*Rd*	*immed12*			
STR | LDR | STRB | LDRB *pre*	*cond*	0 1 0 1	*U op W op*		*Rn*	*Rd*	*immed12*			
STR | LDR | STRB | LDRB *post*	*cond*	0 1 1 0	*U op T op*		*Rn*	*Rd*	*shift_size*	*shift*	0	*Rm*
{ |S|Q|SH| |U|UQ|UH\}ADD16	*cond*	0 1 1 0	0 *op*		*Rn*	*Rd*	1 1 1 1	0 0 0	1	*Rm*
{ |S|Q|SH| |U|UQ|UH\}ADDSUBX	*cond*	0 1 1 0	0 *op*		*Rn*	*Rd*	1 1 1 1	0 0 1	1	*Rm*
{ |S|Q|SH| |U|UQ|UH\}SUBADDX	*cond*	0 1 1 0	0 *op*		*Rn*	*Rd*	1 1 1 1	0 1 0	1	*Rm*
{ |S|Q|SH| |U|UQ|UH\}SUB16	*cond*	0 1 1 0	0 *op*		*Rn*	*Rd*	1 1 1 1	0 1 1	1	*Rm*
{ |S|Q|SH| |U|UQ|UH\}ADD8	*cond*	0 1 1 0	0 *op*		*Rn*	*Rd*	1 1 1 1	1 0 0	1	*Rm*
{ |S|Q|SH| |U|UQ|UH\}SUB8	*cond*	0 1 1 0	0 *op*		*Rn*	*Rd*	1 1 1 1	1 1 1	1	*Rm*
PKHBT | PKHTB	*cond*	0 1 1 0	1 0 0	0	*Rn*	*Rd*	*shift_size*	*op* 0	1	*Rm*
{S|U}SAT	*cond*	0 1 1 0	1 *op* 1		*immed5*	*Rd*	*shift_size*	*sh* 0	1	*Rm*
{S|U}SAT16	*cond*	0 1 1 0	1 *op* 1	0	*immed4*	*Rd*	1 1 1 1	0 0 1	1	*Rm*
SEL	*cond*	0 1 1 0	1 0 0	0	*Rn*	*Rd*	1 1 1 1	1 0 1	1	*Rm*
REV | REV16 | | REVSH	*cond*	0 1 1 0	1 *op* 1	1	1 1 1 1	*Rd*	1 1 1 1	*op* 0 1	1	*Rm*
{S|U}XTAB16	*cond*	0 1 1 0	1 *op* 0	0	*Rn!*=1111	*Rd*	*rot* 0 0	0 1 1	1	*Rm*
{S|U}XTB16	*cond*	0 1 1 0	1 *op* 0	0	1 1 1 1	*Rd*	*rot* 0 0	0 1 1	1	*Rm*
{S|U}XTAB	*cond*	0 1 1 0	1 *op* 1	0	*Rn!*=1111	*Rd*	*rot* 0 0	0 1 1	1	*Rm*
{S|U}XTB	*cond*	0 1 1 0	1 *op* 1	0	1 1 1 1	*Rd*	*rot* 0 0	0 1 1	1	*Rm*
{S|U}XTAH	*cond*	0 1 1 0	1 *op* 1	1	*Rn!*=1111	*Rd*	*rot* 0 0	0 1 1	1	*Rm*
{S|U}XTH	*cond*	0 1 1 0	1 *op* 1	1	1 1 1 1	*Rd*	*rot* 0 0	0 1 1	1	*Rm*
STR | LDR | STRB | LDRB *pre*	*cond*	0 1 1 1	*U op W op*		*Rn*	*Rd*	*shift_size*	*shift*	0	*Rm*
SMLAD | SMLSD	*cond*	0 1 1 1	0 0 0	0	*Rd*	*Rn!*=1111	*Rs*	0 *op X*	1	*Rm*
SMUAD | SMUSD	*cond*	0 1 1 1	0 0 0	0	*Rd*	1 1 1 1	*Rs*	0 *op X*	1	*Rm*
SMLALD | SMLSLD	*cond*	0 1 1 1	0 1 0	0	*RdHi*	*RdLo*	*Rs*	0 *op X*	1	*Rm*

Virtual machines

Abstractions for computers




```
A       DUP   32

        lda   R1, 1
        lda   RA, A
        lda   RC, 0

read    ld    RD, 0xFF
        bz    RD, exit
        add   R2, RA, RC
        sti   RD, R2
        add   RC, RC, R1
        bz    R0, read

exit    jl    RF, printr
        hlt
```

TOY assembly

TOY assembly

Not mapping to instruction

- Data directives
- A DW n: initialize a variable A as n
- B DUP n: reserve n words (n is decimal)
- Support two types of literals, decimal and hexadecimal (0x)
- Label begins with a letter
- Comment begins with ;
- Case insensitive
- Program starts with the first instruction it meets

- Some tricks to handle the starting address 0x10

opcode	mnemonic	syntax
0	hlt	hlt
1	add	add rd, rs, rt
2	sub	sub rd, rs, rt
3	and	and rd, rs, rt
4	xor	xor rd, rs, rt
5	shl	shl rd, rs, rt
6	shr	shr rd, rs, rt
7	lda	lda rd, addr
8	ld	ld rd, addr
9	st	st rd, addr
A	ldi	ldi rd, rt
B	sti	sti rd, rt
C	bz	bz rd, addr
D	bp	bp rd, addr
E	jr	jr rd (rt)
F	jl	jl rd, addr

Assembler

Assembler's task:

- Convert mnemonic operation codes to their machine language equivalents
- Convert symbolic operands to their equivalent machine addresses
- Build machine instructions in proper format
- Convert data constants into internal machine representations (data formats)
- Write object program and the assembly listing

Forward Reference

Definition

- A reference to a label that is defined later in the program

Solution

- Two passes
  - First pass: scan the source program for label definition, address accumulation, and address assignment
  - Second pass: perform most of the actual instruction translation

Assembly version of REVERSE

```
int A[32];          A       DUP   32                10: C020

                            lda   R1, 1             20: 7101
                            lda   RA, A             21: 7A00
i=0;                        lda   RC, 0             22: 7C00
Do {
  RD=stdin;         read    ld    RD, 0xFF          23: 8DFF
  if (RD==0) break;         bz    RD, exit          24: CD29
                            add   R2, RA, RC        25: 12AC
  A[i]=RD;                  sti   RD, R2            26: BD02
  i=i+1;                    add   RC, RC, R1        27: 1CC1
} while (1);                bz    R0, read          28: C023

printr();           exit    jl    RF, printr        29: FF2B
                            hlt                     2A: 0000
```

Assembly version of REVERSE

```
printr()                ; print reverse
{                       ; array address (RA)
  do {                  ; number of elements (RC)
    i=i-1;              printr  sub     RC, RC, R1      2B: 2CC1
                                add     R2, RA, RC      2C: 12AC
                                ldi     RD, R2          2D: AD02
    print A[i];                 st      RD, 0xFF        2E: 9DFF
  } while (i>=0);               bp      RC, printr      2F: DC2B
                                bz      RC, printr      30: CC2B
  return;               return  jr      RF              31: EF00
}
```

toyasm < reverse.asm > reverse.toy

Function Call: A Failed Attempt

Goal: $x \times y \times z$.

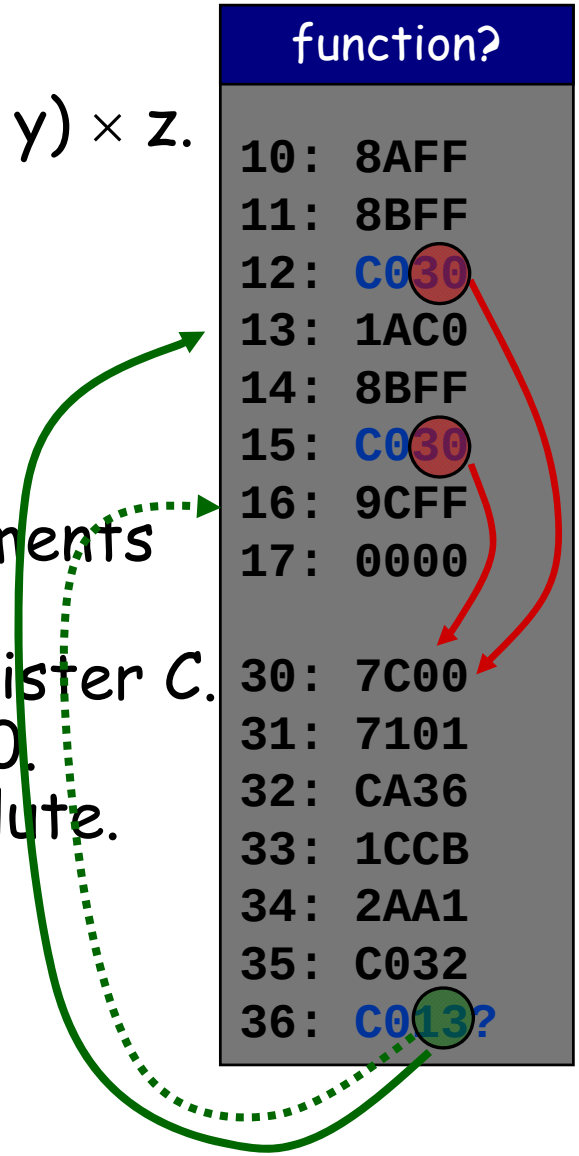
- Need two multiplications: $x \times y$, $(x \times y) \times z$.
  - Solution 1: write multiply code 2 times.
  - Solution 2: write a TOY function.

A failed attempt:

- Write multiply loop at 30-36.
- Calling program agrees to store arguments in registers A and B.
- Function agrees to leave result in register C.
- Call function with jump absolute to 30.
- Return from function with jump absolute.

Reason for failure.

- Need to return to a VARIABLE memory address.



Multiplication Function

Calling convention.

- Jump to line 30.
- Store a and b in registers A and B.
- Return address in register F.
- Put result c = a × b in register C.
- Register 1 is scratch.
- Overwrites registers A and B.

function.toy

```
30: 7C00   R[C] ← 00
31: 7101   R[1] ← 01
32: CA36   if (R[A] == 0) goto 36
33: 1CCB   R[C] += R[B]
34: 2AA1   R[A]--
35: C032   goto 32
36: EF00   pc ← R[F]
```

return

opcode E
jump register

function

```
10: 8AFF
11: 8BFF
12: FF30
13: 1AC0
14: 8BFF
15: FF30
16: 9CFF
17: 0000

30: 7C00
31: 7101
32: CA36
33: 1CCB
34: 2AA1
35: C032
36: EF00
```

Multiplication Function Call

Client program to compute x × y × z.

- Read x, y, z from standard input.
- Note: PC is incremented before instruction is executed.
  - value stored in register F is correct return address

function.toy (cont)

```
10: 8AFF   read R[A]                x
11: 8BFF   read R[B]                y
12: FF30   R[F] ← pc; goto 30       x * y
13: 1AC0   R[A] ← R[C]              (x * y)
14: 8BFF   read R[B]                z
15: FF30   R[F] ← pc; goto 30       (x * y) * z
16: 9CFF   write R[C]
17: 0000   halt
```

opcode F
jump and link

R[F] ← 13

R[F] ← 16

Function Call: One Solution

Contract between calling program and function:

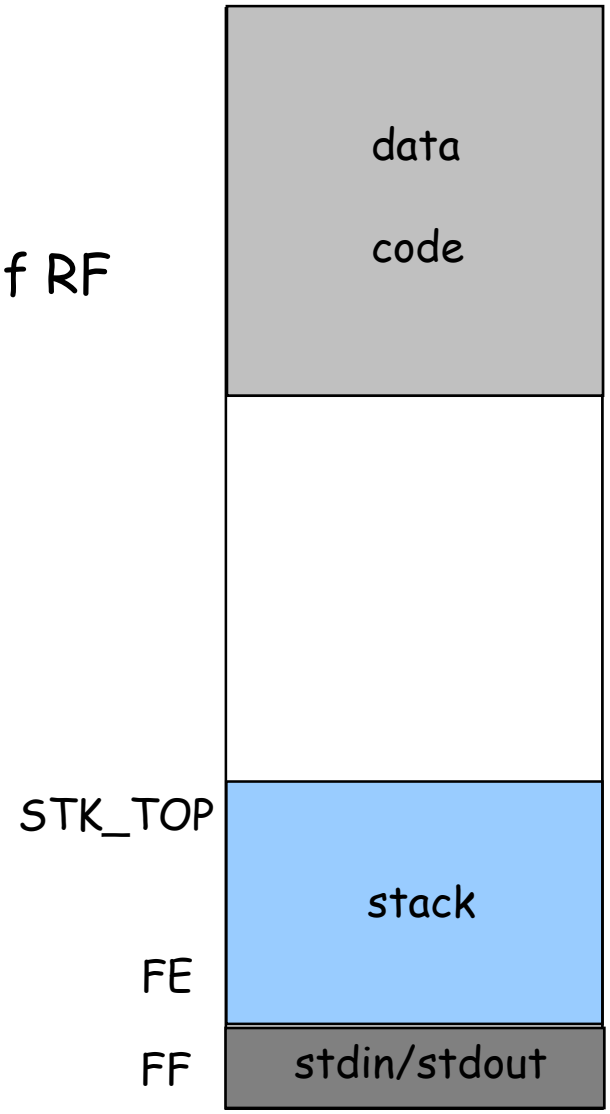
- Calling program stores function parameters in specific registers.
- Calling program stores return address in a specific register.
  - jump-and-link
- Calling program sets PC to address of function.
- Function stores return value in specific register.
- Function sets PC to return address when finished.
  - jump register

What if you want a function to call another function?

- Use a different register for return address.
- More general: store return addresses on a stack.

stack

```
STK_TOP     DW      0xFF

; these procedures will use R8, R9
; assume return address is in RE, instead of RF
; it is the only exception

; push RF into stack
push    lda     R8, 1
        ld      R9, STK_TOP
        sub     R9, R9, R8
        st      R9, STK_TOP
        sti     RF, R9
        jr      RE
```

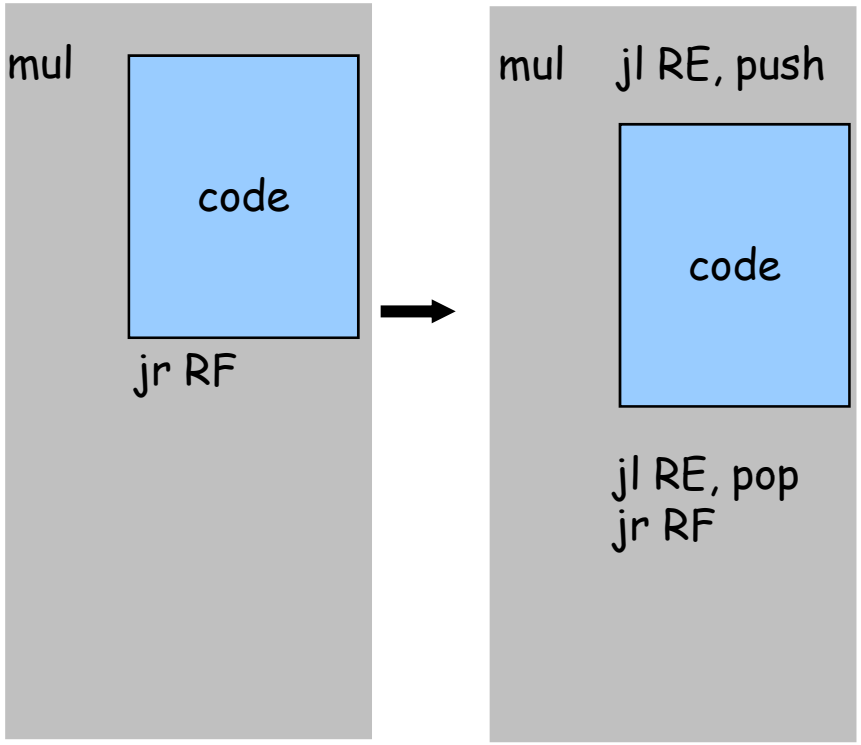


stack

```
; pop and return [top] to RF
pop     lda     R8, 0xFF
        ld      R9, STK_TOP
        sub     R8, R8, R9
        bz      R8, popexit
        ldi     RF, R9
        lda     R8, 1
        add     R9, R9, R8
        st      R9, STK_TOP
popexit jr      RE

; the size of the stack, the result is in R9
stksize lda     R8, 0xFF
        ld      R9, STK_TOP
        sub     R9, R8, R9
        jr      RE
```

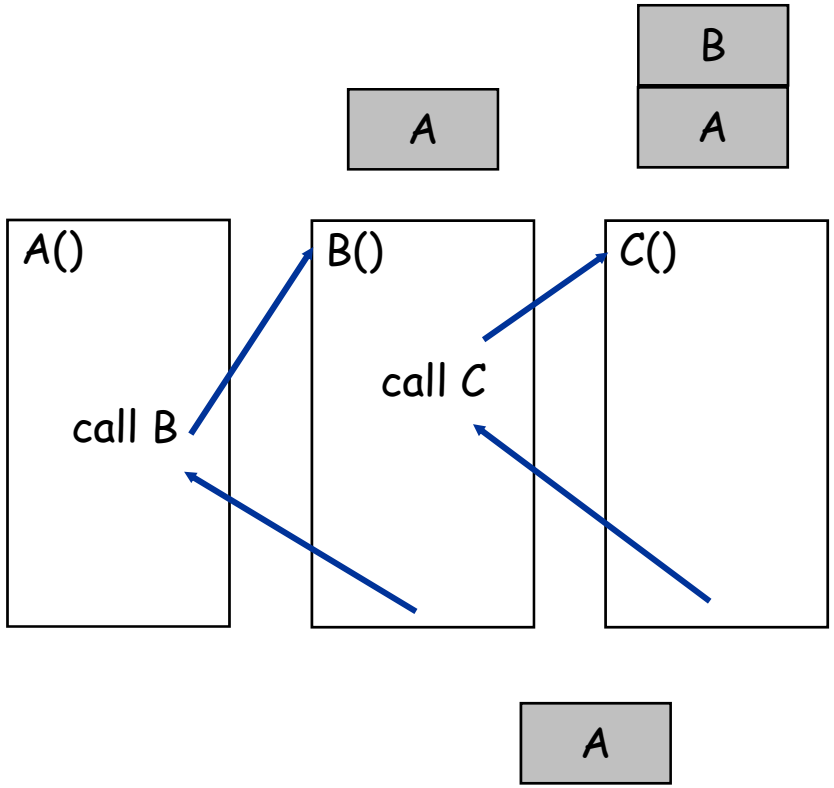
Procedure prototype

With a stack, the procedure prototype is changed. It allows us to have a deeper call graph by using the stack.

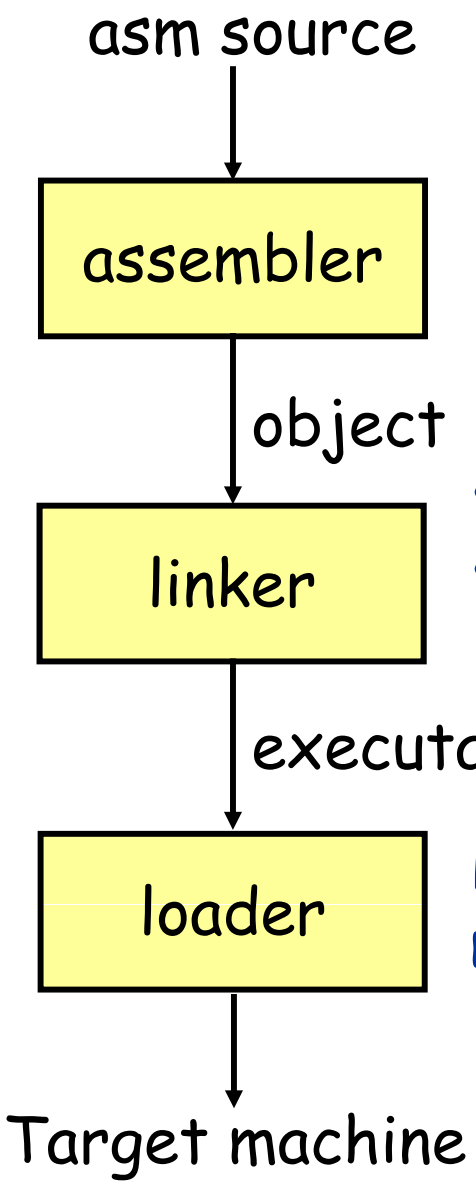


before

after



Assembly programming flow



- Combine separate object codes
- Supply necessary information for references between them

Bring the object program into memory for execution

Linking

Many programs will need multiply. Since multiply will be used by many applications, could we make multiply a library?

Toyasm has an option to generate an object file so that it can be later linked with other object files.

That is why we need linking. Write a subroutine mul3 which multiplies three numbers in RA, RB, RC together and place the result in RD.
Three files:

- stack.obj: implementation of stack, needed for procedure
- mul.obj: implementation of multiplication.
- multest.obj: main program and procedure of mul3

```
toylink multest.obj mul.obj stack.obj > multest.toy
```

object file (multest.asm)

```
A       DW      3
B       DW      4
C       DW      5

; calculate A*B*C
main    ld      RA, A
        ld      RB, B
        ld      RC, C
        jl      RF, mul3
        st      RD, 0xFF
        hlt

; RD=RA*RB*RC
; return address is in RF
mul3    jl      RE, push

        lda     RD, 0
        add     RD, RC, R0
        jl      RF, mul
        add     RA, RC, R0
        add     RB, RD, R0
        jl      RF, mul
        add     RD, RC, R0

        jl      RE, pop
        jr      RF
```

object file (mul.obj)

```
SIXTEEN      DW     16

; multiply RC=RA*RB
; return address is in RF
; it will modify R2, R3 and R4 as well
mul      jl       RE, push

         lda      RC, 0
         lda      R1, 1
         ld       R2, SIXTEEN
m_loop   sub      R2, R2, R1
         shl      R3, RA, R2
         shr      R4, RB, R2
         and      R4, R4, R1
         bz       R4, m_end
         add      RC, RC, R3
m_end    bp       R2, m_loop

         jl       RE, pop
         jr       RF
```

These are literals.
No need to relocate

```
// size 29
// export 4
// SIXTEEN 0x00
// mul 0x10
// m_loop 0x14
// m_end 0x1A
// literal 2 17 18
// lines 14
00: 0010
10: FE00
11: 7000
12: 7101
13: 8200
14: 2221
15: 53A2
16: 64B2
17: 3441
18: C41A
19: 1CC3
1A: D214
1B: FE00
1C: EF00
// import 2
// push 1 0x10
// pop 1 0x1B
```

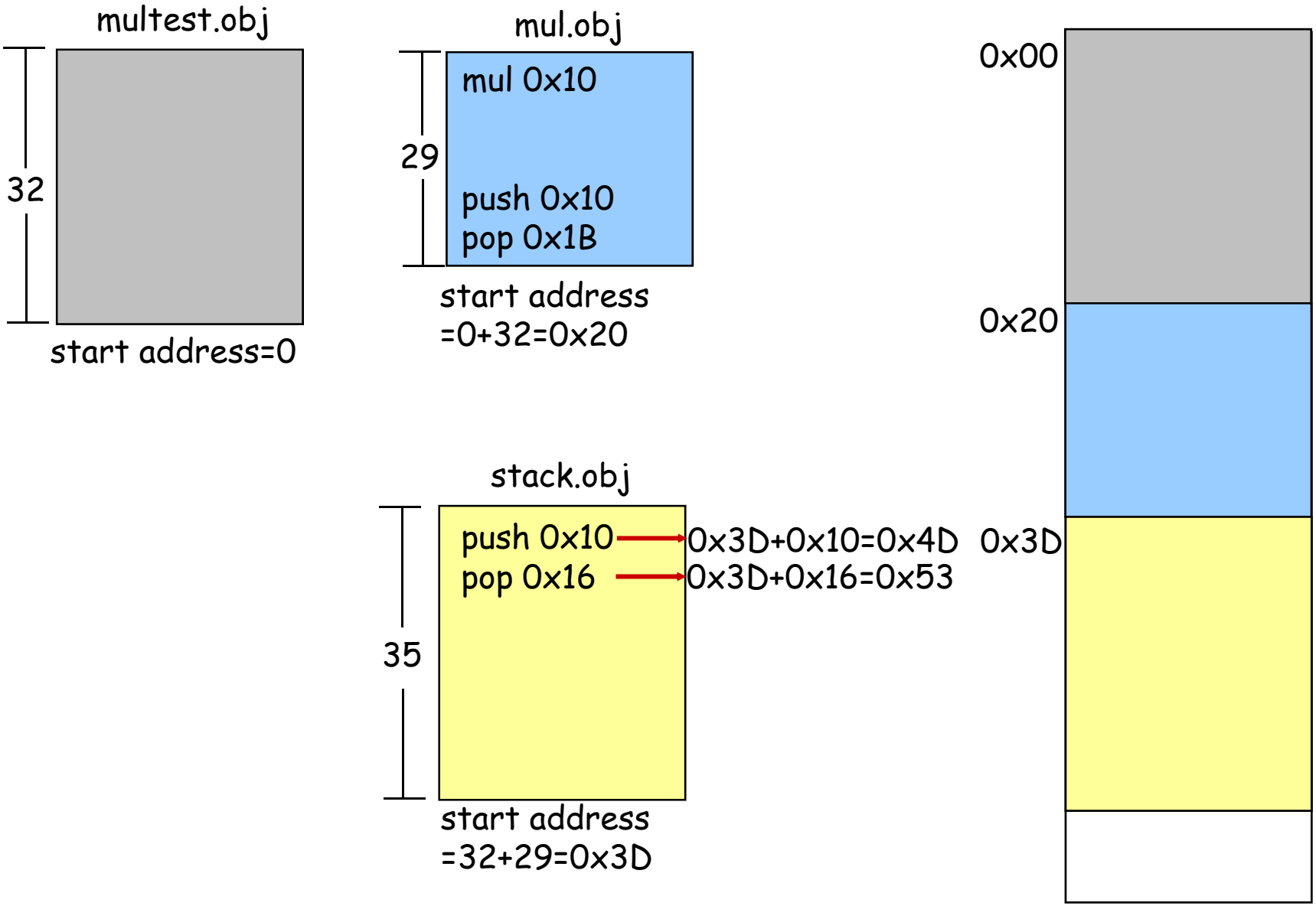
export table

import table

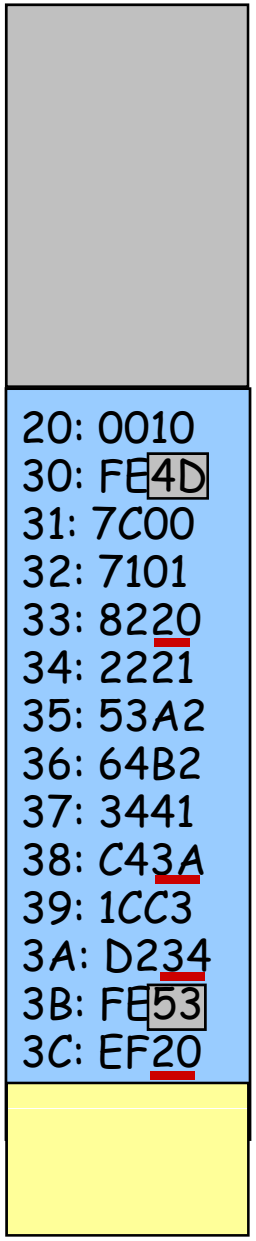
need to fill in address of push once we know it

need to fill in address of pop once we know it

Linking


multest.obj
32
start address=0
mul.obj
29
mul 0x10
push 0x10
pop 0x1B
start address
=0+32=0x20
stack.obj
35
push 0x10
pop 0x16
0x3D+0x10=0x4D
0x3D+0x16=0x53
start address
=32+29=0x3D
0x00
0x20
0x3D

Resolve external symbols



export table

```
// size 29
// export 4
// SIXTEEN 0x00
// mul 0x10
// m_loop 0x14
// m_end 0x1A
// literal 2 17 18
// lines 14
00: 0010
10: FE00
11: 7C00
12: 7101
13: 8200
14: 2221
15: 53A2
16: 64B2
17: 3441
18: C41A
19: 1CC3
1A: D214
1B: FE00
1C: EF00
```

These are literals.
No need to relocate

need to fill in address of push once we know it

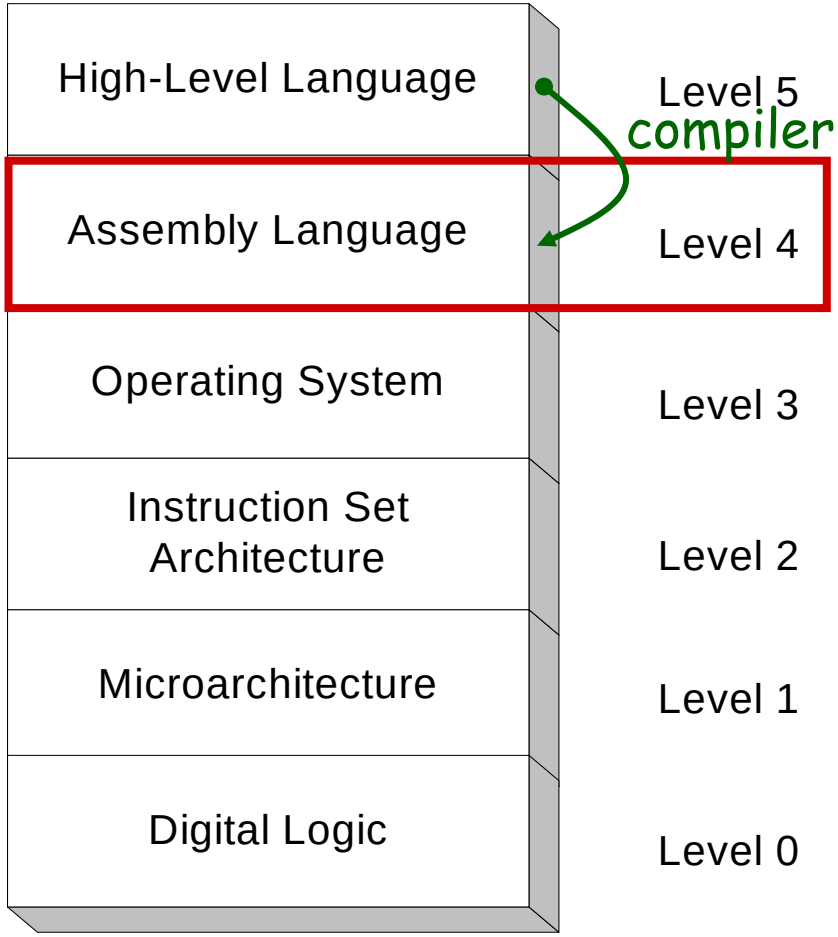
need to fill in address of pop once we know it

import table

```
// import 2
// push 1 0x10
// pop 1 0x1B
```

Virtual machines

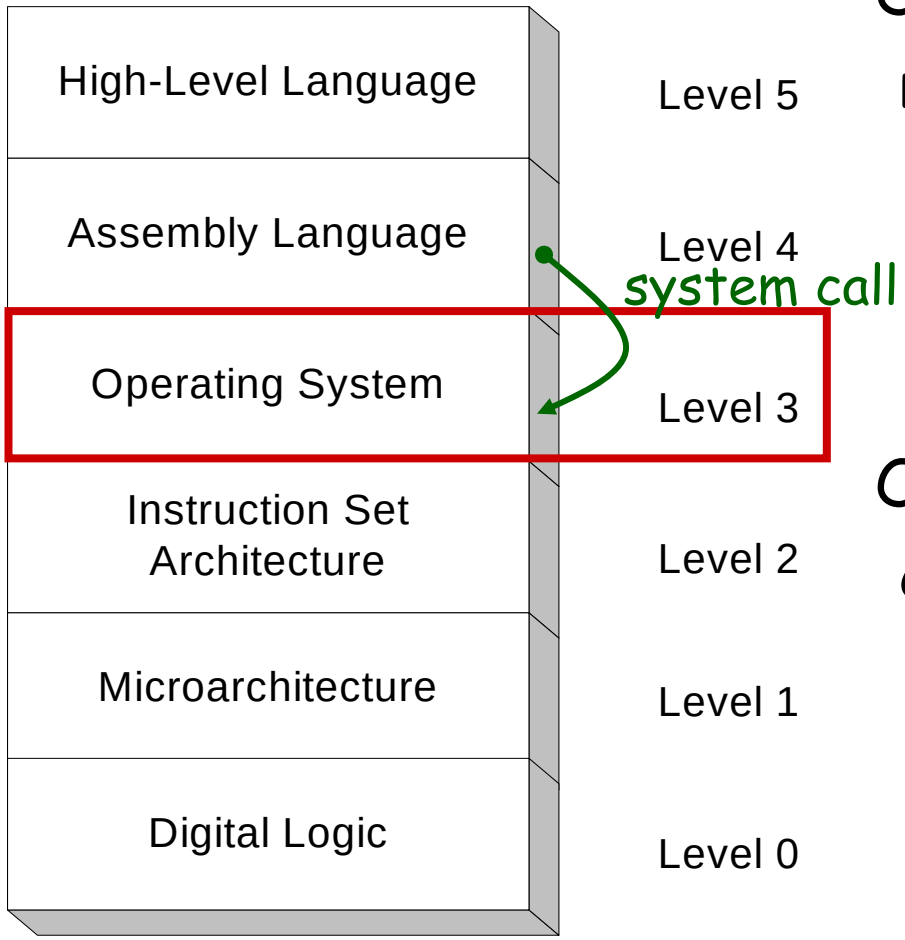
Abstractions for computers




```
A       DUP     32

        lda     R1, 1
        lda     RA, A
        lda     RC, 0

read    ld      RD, 0xFF
        bz      RD, exit
        add     R2, RA, RC
        sti     RD, R2
        add     RC, RC, R1
        bz      R0, read

exit    jl      RF, printr
        hlt
```

Virtual machines

Abstractions for computers



Operating system is a resource allocator

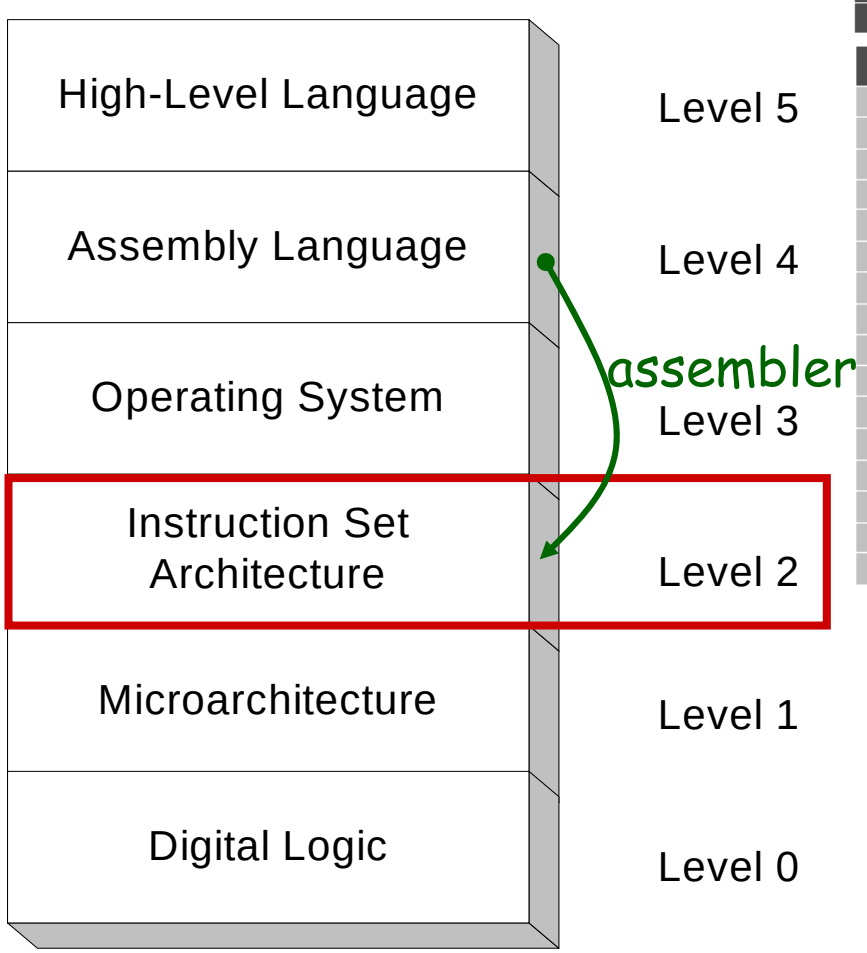
- Managing all resources (memory, I/O, execution)
- Resolving requests for efficient and fair usage

Operating system is a control program

- Controlling execution of programs to prevent errors and improper use of the computer

Virtual machines

Abstractions for computers



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Format 1	opcode				dest d				source s				source t			
Format 2	opcode				dest d				addr							

#	Operation	Emt	Pseudocode
0:	halt	1	exit(0)
1:	add	1	R[d] ← R[s] + R[t]
2:	subtract	1	R[d] ← R[s] - R[t]
3:	and	1	R[d] ← R[s] & R[t]
4:	xor	1	R[d] ← R[s] ^ R[t]
5:	shift left	1	R[d] ← R[s] << R[t]
6:	shift right	1	R[d] ← R[s] >> R[t]
7:	load addr	2	R[d] ← addr
8:	load	2	R[d] ← mem[addr]
9:	store	2	mem[addr] ← R[d]
A:	load indirect	1	R[d] ← mem[R[t]]
B:	store indirect	1	mem[R[t]] ← R[d]
C:	branch zero	2	if (R[d] == 0) pc ← addr
D:	branch positive	2	if (R[d] > 0) pc ← addr
E:	jump register	1	pc ← R[t]
F:	jump and link	2	R[d] ← pc; pc ← addr

10: C020

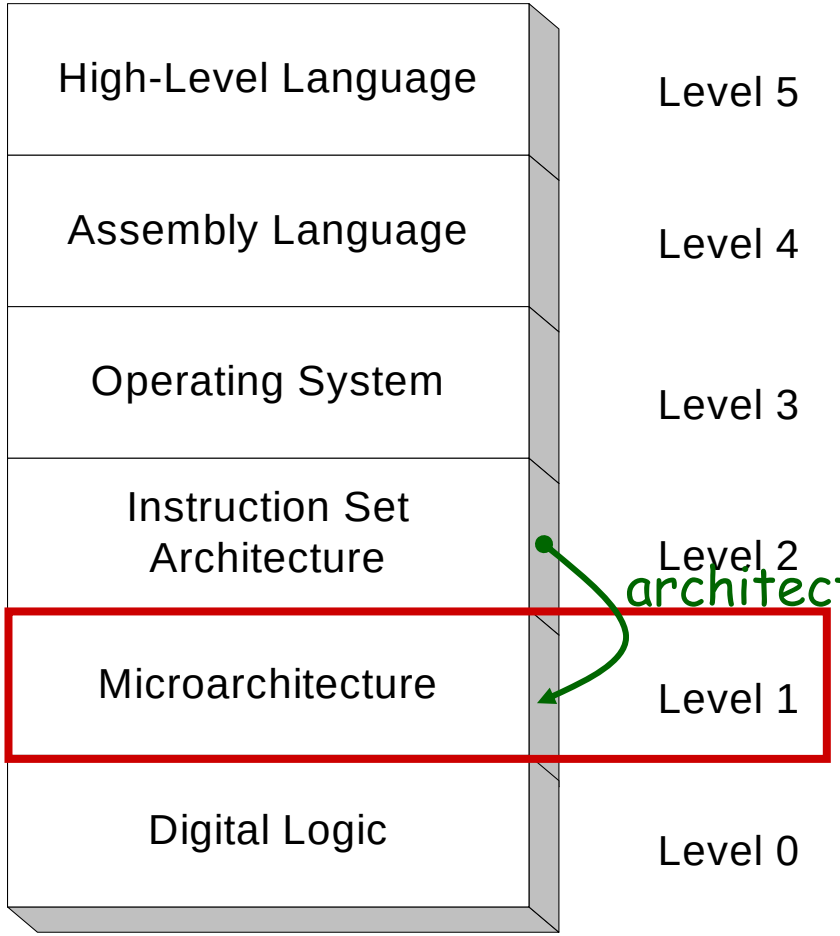
20: 7101
21: 7A00
22: 7C00

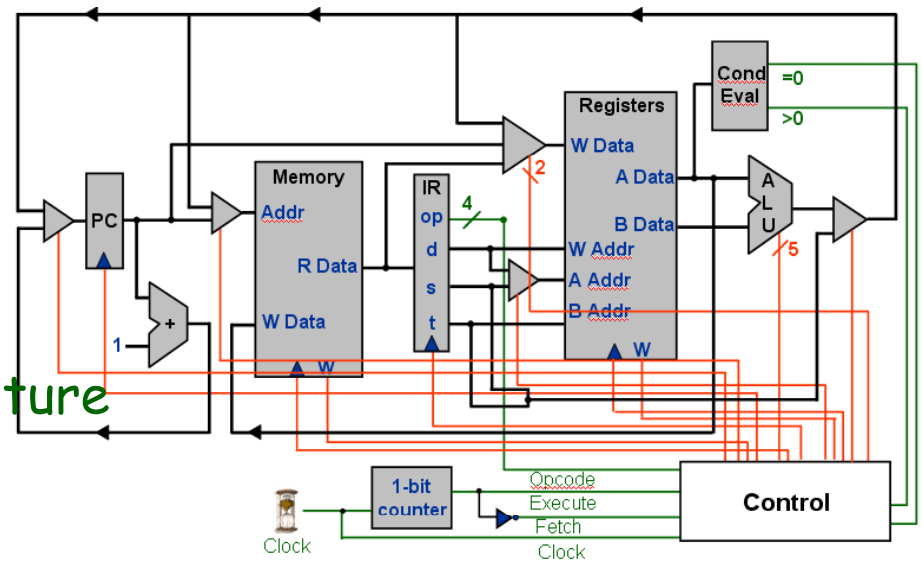
23: 8DFF
24: CD29
25: 12AC
26: BD02
27: 1CC1
28: C023

29: FF2B
2A: 0000

Virtual machines

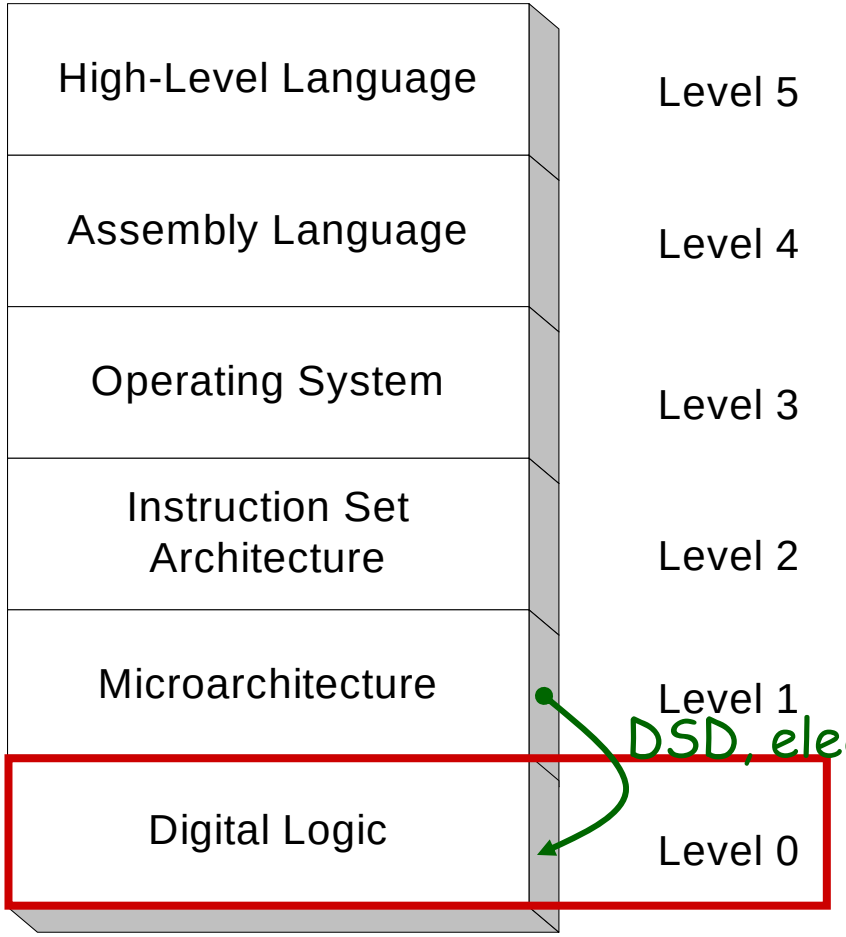
Abstractions for computers

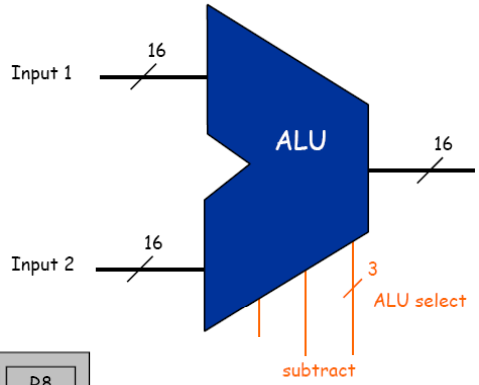


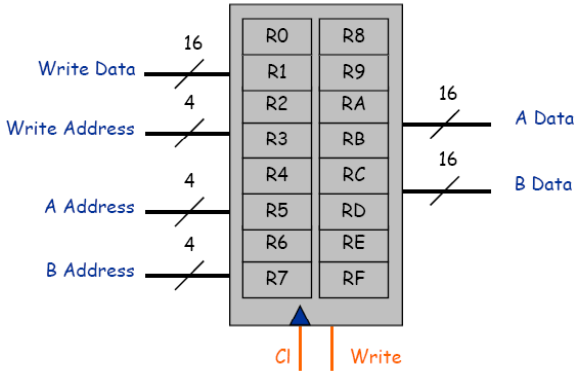


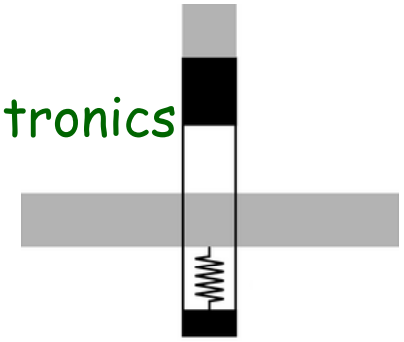
Virtual machines

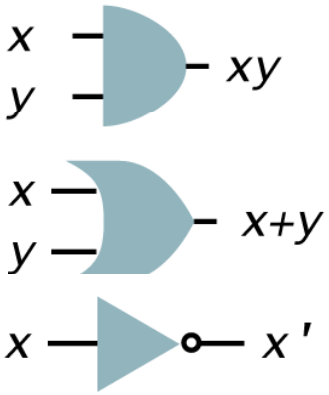
Abstractions for computers











Assignment #2

Assigned: 11/03/2008
Due: 11:59pm 11/16/2008
Part 1 (50%): write a procedure BCD to convert a hexadecimal number into a BCD (Binary-Coded Decimal). The input number is placed in RA. The result should be placed in RB. The return address is in RF. (Hint: you need to implement division)

Part 2 (30%): write a procedure CNT0 to count 0's in an array. The address of the array is placed at RA. The size of the array is specified by RC. The result should be placed in RB. The return address is in RF.

Part 3 (20%): write a program to read a series of numbers specified by the user from stdin until the input is 0x0000. Count the number of 0-bits in the input array and display this number using BCD in stdout.