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
```

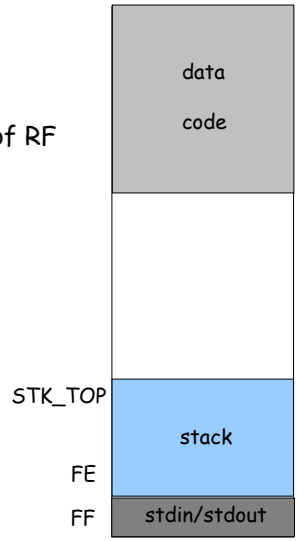

Assembly version of REVERSE

```
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

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
```

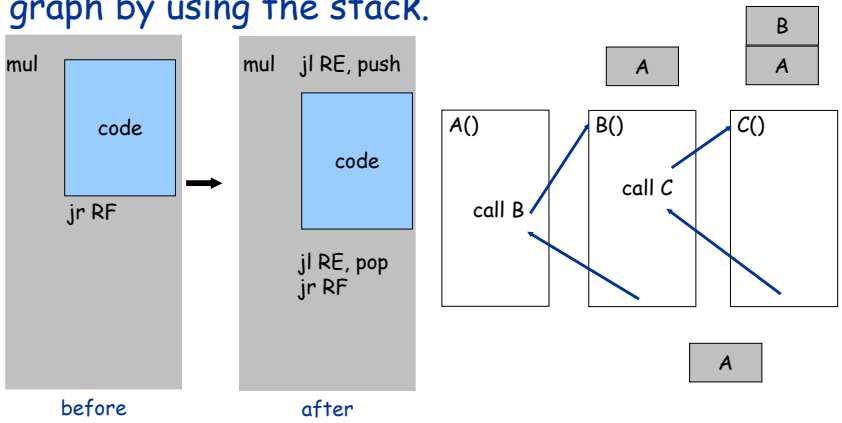


stack

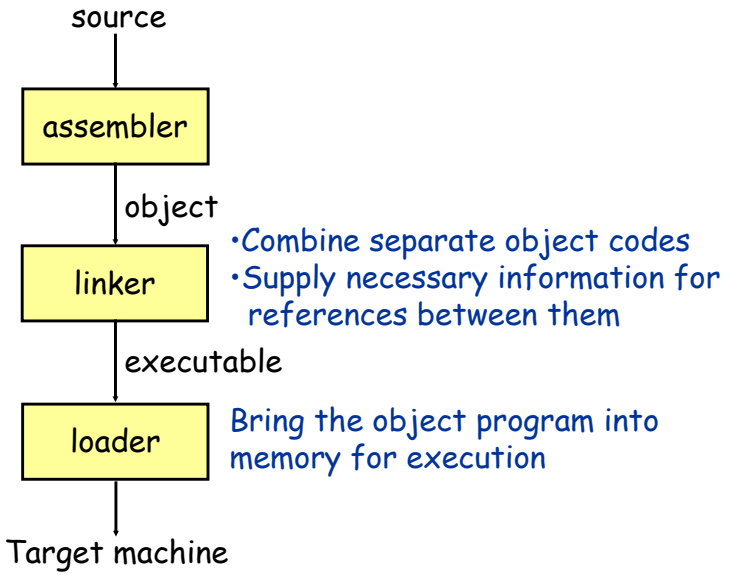
```
; pop and return [top] to RF
pop     lda     R8, 0xFF
        ld      R9, STK_TOP
        sub     R8, R8, R9
        bz      R8, popexit
        ldi     RF, R9
        lda     R8, 1
        add     R9, R9, R8
        st      R9, STK_TOP
popexit jr      RE

; 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.



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.obj)

```
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

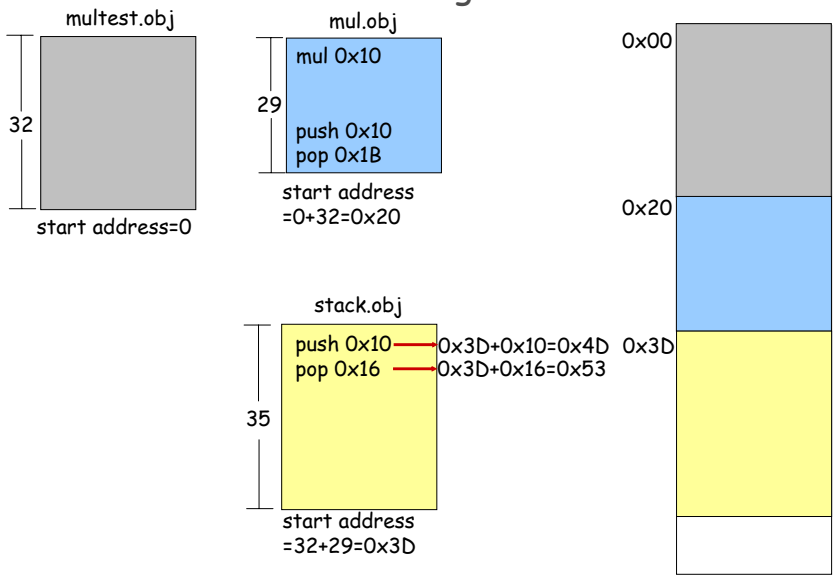
export table

```
// size 29
// export 4
// SIXTEEN 0x00
// mul 0x10
// m_loop 0x14
// m_end 0x1A
// literal 2 17 18
// lines 14
00: 0010
10: FE00   <- need to fill in address of push once we know it
11: 7C00
12: 7101
13: 8200
14: 2221
15: 53A2
16: 64B2
17: 3441
18: C41A
19: 1CC3
1A: D214
1B: FE00   <- need to fill in address of pop once we know it
1C: EF00
```

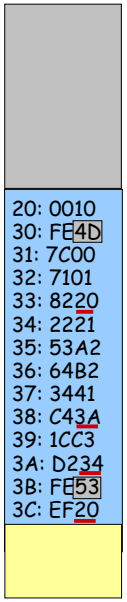
import table

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


Linking



Resolve external symbols



These are literals. No need to relocate

export table

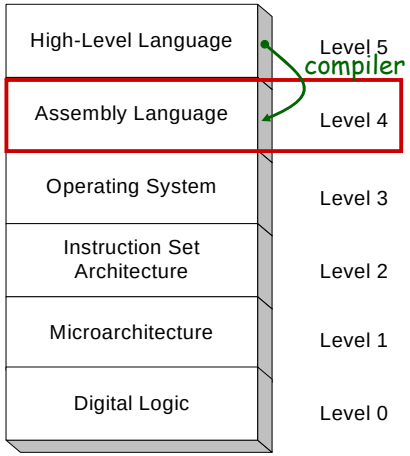
```
// size 29
// export 4
// SIXTEEN 0x00
// mul 0x10
// m_loop 0x14
// m_end 0x1A
// literal 2 17 18
// lines 14
00: 0010
10: FE00   <- need to fill in address of push once we know it
11: 7C00
12: 7101
13: 8200
14: 2221
15: 53A2
16: 64B2
17: 3441
18: C41A
19: 1CC3
1A: D214
1B: FE00   <- need to fill in address of pop once we know it
1C: EF00
```

import table

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


Virtual machines

Abstractions for computers



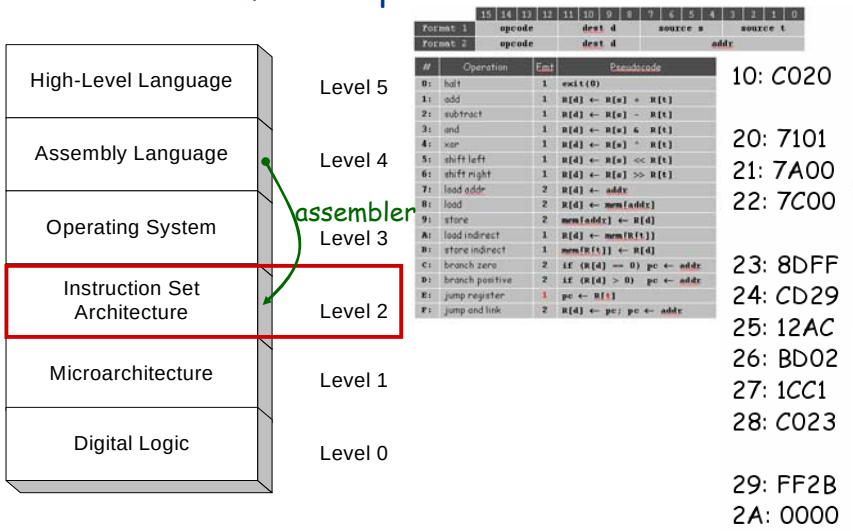

```
A       DUP   32

        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
```

Operating System

- 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

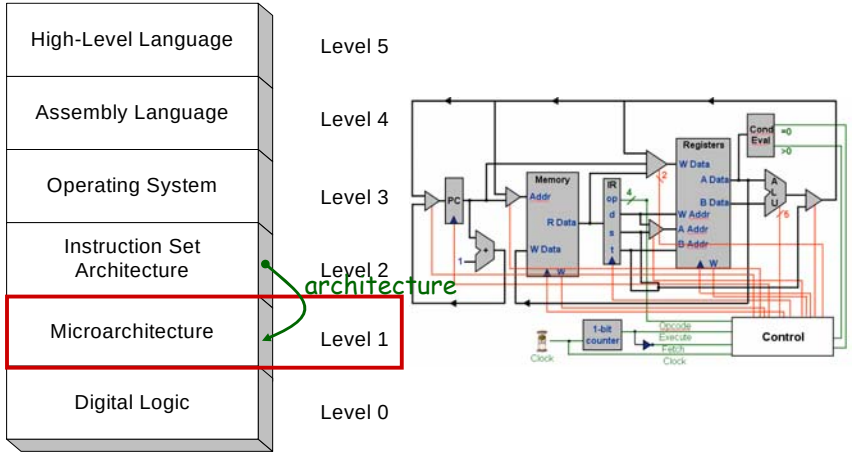
Virtual machines

Abstractions for computers



Virtual machines

Abstractions for computers



Virtual machines

Abstractions for computers