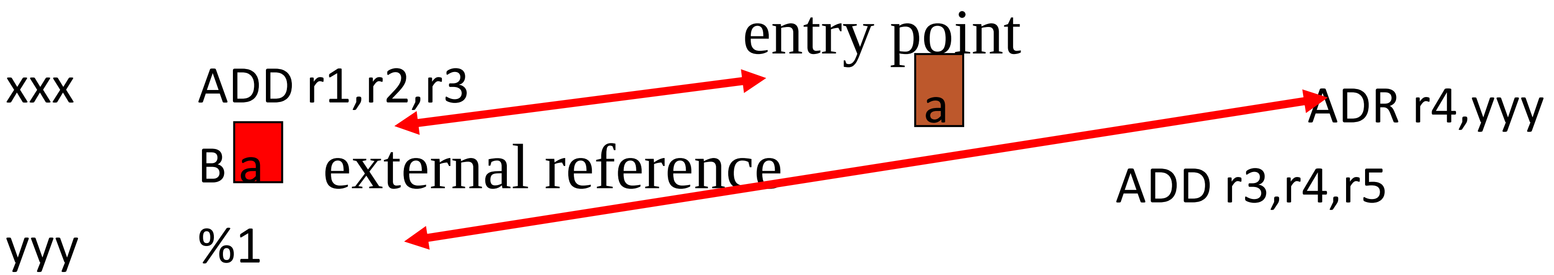
Linking

Combines several object modules into a single executable module.

Jobs:

- put modules in order;
- resolve labels across modules.

Externals and entry points



Example

```
label1   LDR r0,[r1]
         ...
         ADR a
         ...
         B label2
         ...
var1     %  1
```

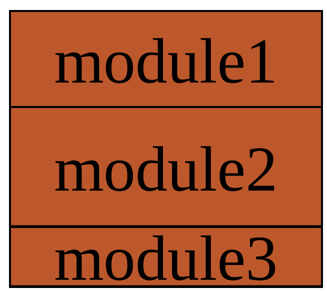
```
label2   ADR var1
         ...
         B label3
         ...
x        %  1
y        %  1
a        %  10
```

External references	Entry points
a	label1
label2	var1

External references	Entry points
var1	label2
label3	x
	y
	a

Module ordering

- Code modules must be placed in absolute positions in the memory space.
- Load map or linker flags control the order of modules.

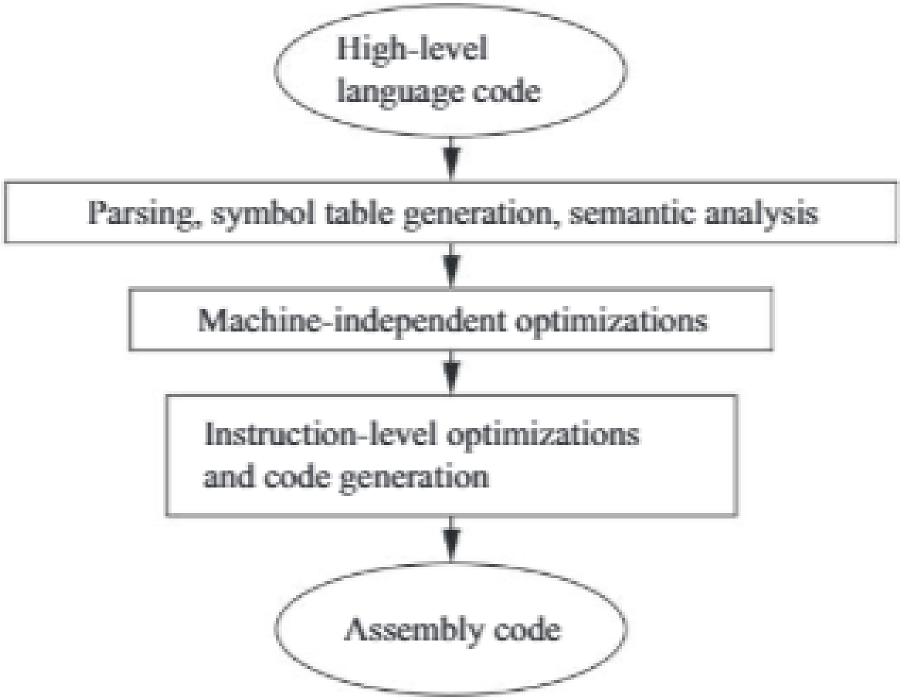


Dynamic linking

Some operating systems link modules dynamically at run time:

- shares one copy of library among all executing programs;
- allows programs to be updated with new versions of libraries.

Compilation process

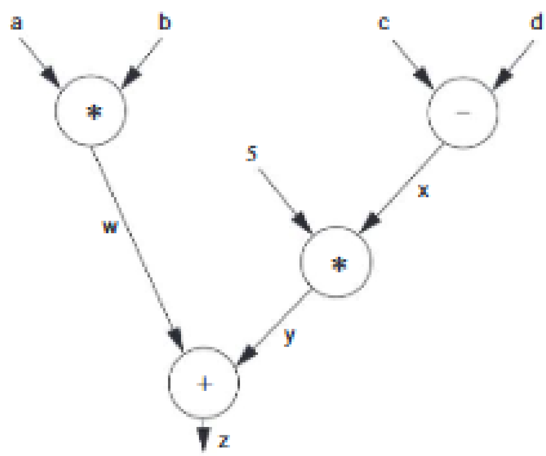


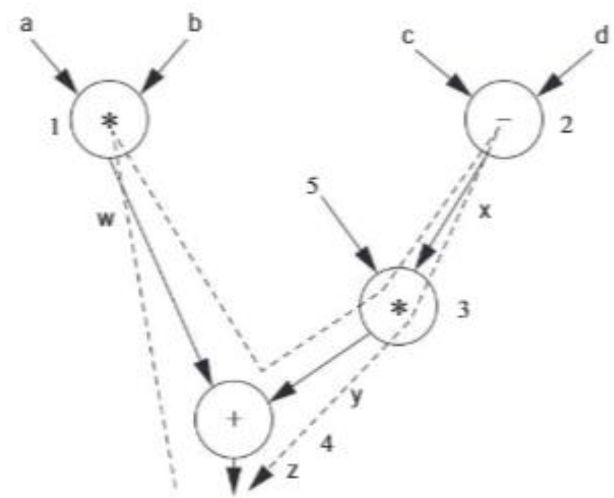
Basic compilation methods

- Support of functions or procedures
- As we have seen in computer architecture:
  - Some times you need to use the stack to backup register files before using them.
- Use stack pointer and frame pointers.
  - Frames are loaded into the stack in order
  - The stack pointer should not exceed the frame pointer.

Arithmetic expressions

$$x = a*b + 5*(c - d)$$



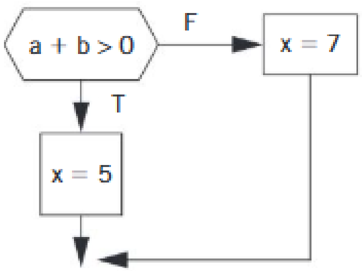


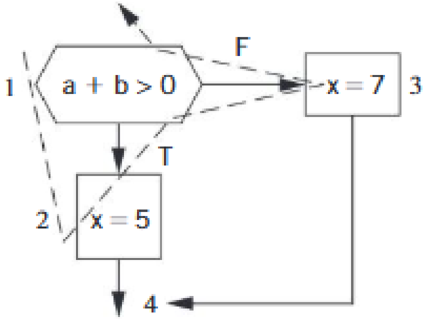
Assembly Code of previous example

```
; operator 1 (*)
ADR r4,a             ; get address for a
MOV r1,[r4]          ; load a
ADR r4,b             ; get address for b
MOV r2,[r4]          ; load b
ADD r3,r1,r2         ; put w into r3
; operator 2 (-)
ADR r4,c             ; get address for c
MOV r4,[r4]          ; load c
ADR r4,d             ; get address for d
MOV r5,[r4]          ; load d
SUB r6,r4,r5         ; put z into r6
; operator 3 (*)
MUL r7,r6,#5         ; operator 3, puts y into r7
; operator 4 (+)
ADD r8,r7,r3         ; operator 4, puts x into r8
; assign to x
ADR r1,x
STR r8,[r1]          ; assigns to x location
```

Conditional operations

```
if (a + b >0)
        x = 5;
else
        x = 7;
```



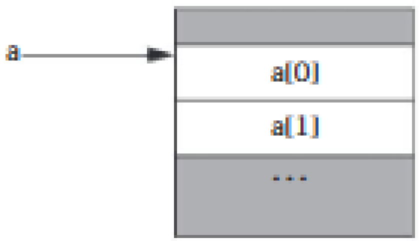


Assembly code of previous example

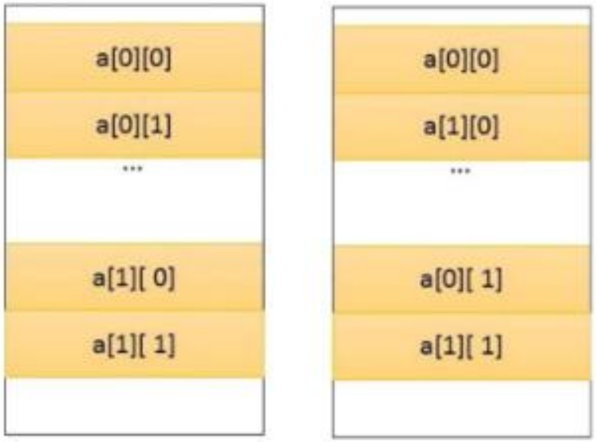
```
        ADR r5,a        ; get address for a
        LDR r1,[r5]     ; load a
        ADR r5,b        ; get address for b
        LDR r2,b        ; load b
        ADD r3,r1,r2
        BLE label3      ; true condition falls through branch
  ; true case
        LDR r3,#5       ; load constant
        ADR r5,x
        STR r3, [r5]    ; store value into x
        B stmtend       ; done with the true case

; false case
label3  LDR r3,#7       ; load constant
        ADR r5,x        ; get address of x
        STR r3,[r5]     ; store value into x
stmtend
```

Arrays



a[i]



row major column major

a[i][j]

a[i*M + j]

Compiler optimization

- Function Inlining:
  - Substitute subroutine call to a function with an equivalent call to the function body.
  - Eliminate the subroutine overhead.

- Function outlining
  - Replace similar sections of the code with a call to an equivalent function
  - Reduce code size

Compiler optimization

- Loop unrolling:

```
for (i = 0; i <N; i++) {
    a[i]=b[i]*c[i];
}
```

```
a[0] = b[0]*c[0];
a[1] = b[1]*c[1];
a[2] = b[2]*c[2];
a[3] = b[3]*c[3];
```

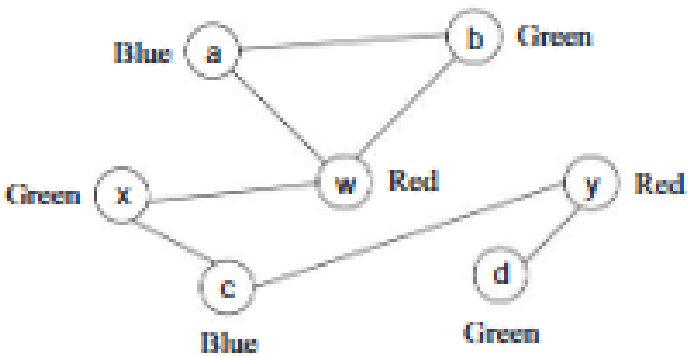
- It may expand the size of the code
- Sometimes compilers can do partial loop unrolling.

Complier optimization

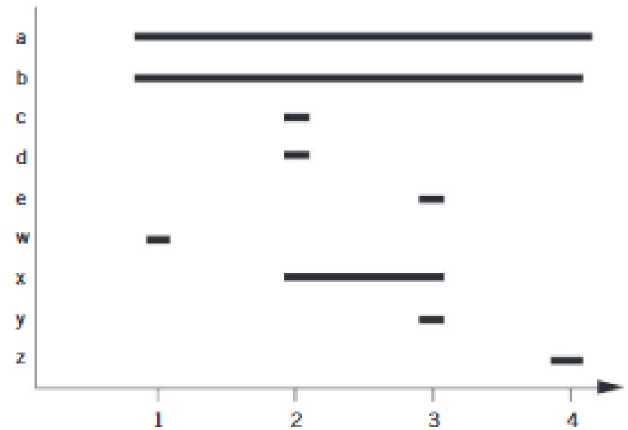
- Loop fusion
- Dead code elimination.
- Register allocation
- Operator scheduling for register allocation

Register allocation example

```
w = a + b; /* statement 1 */
x = c + d; /* statement 2 */
y = x + e; /* statement 3 */
z = a - b; /* statement 4 */
```



Conflict graph



a	r0
b	r1
c	r2
d	r0
w	r3
x	r0
y	r3

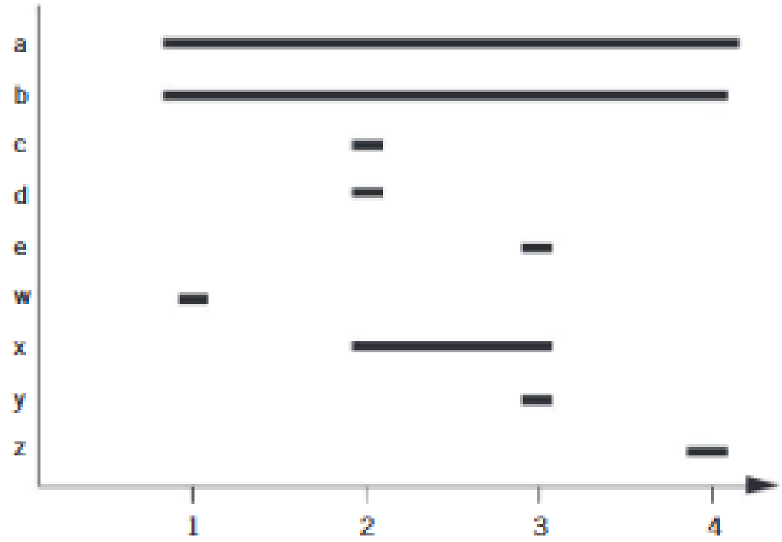
Assembly code for previous example

```
LDR r0,[p_a]     ; load a into r0 using pointer to a (p_a)
LDR r1,[p_b]     ; load b into r1
ADD r3,r0,r1     ; compute a + b
STR r3,[p_w]     ; w = a + b
LDR r2,[p_c]     ; load c into r2
ADD r0,r2,r3     ; compute c + w, reusing r0 for x
STR r0,[p_x]     ; x = c + w
LDR r0,[p_d]     ; load d into r0
ADD r3,r2,r0     ; compute c + d, reusing r3 for y
STR r3,[p_y]     ; y = c + d
```

Rescheduling operations example

```
w = a + b; /* statement 1 */
x = c + d; /* statement 2 */
y = x + e; /* statement 3 */
z = a – b; /* statement 4 */
```

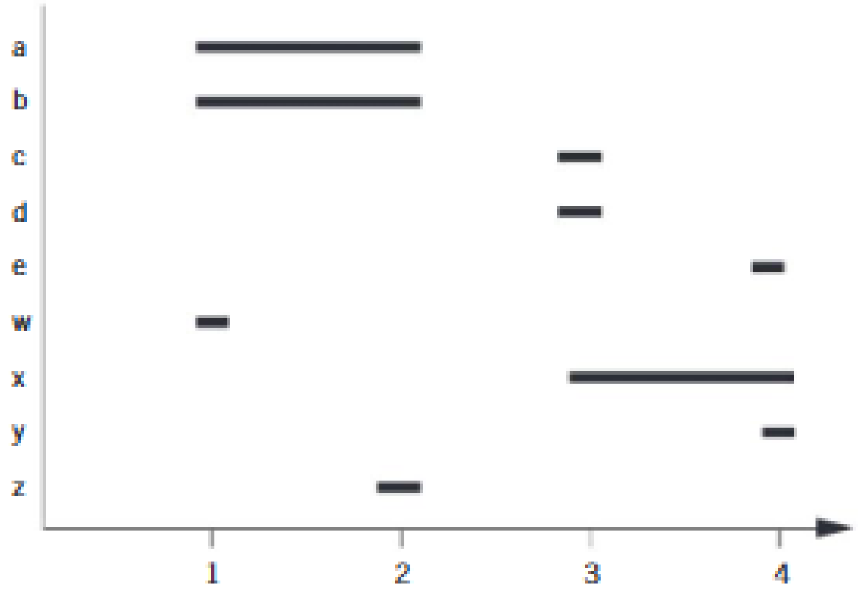
If we compile the statements in the order in which they were written, we get this register graph:



Rescheduling operations example

```
w = a + b; /*statement 1 */
z = a – b; /* statement 29 */
x = c + d; /*statement 39 */
y = x + e; /*statement 49 */
```

Additionally, here is the lifetime graph for the new code:



Two versions of assembly code

Before version

```
LDR r0,a
LDR r1,b
ADD r2,r0,r1
STR r2,w ; w = a + b
LDRr r0,c
LDR r1,d
ADD r2,r0,r1
STR r2,x ; x = c + d
LDR r1,e
ADD r0,r1,r2
STR r0,y ; y = x + e
LDR r0,a ; reload a
LDR r1,b ; reload b
SUB r2,r1,r0
STR r2,z ; z = a – b
```

After version

```
LDR r0,a
LDR r1,b
ADD r2,r1,r0
STR r2,w ; w = a + b
SUB r2,r0,r1
STR r2,z ; z = a – b
LDR r0,c
LDR r1,d
ADD r2,r1,r0
STR r2,x ; x = c + d
LDR r1,e
ADD r0,r1,r2
STR r0,y ; y = x + e
```