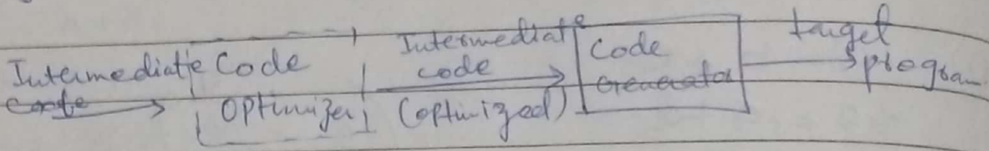
CODE GENERATION

Date 12-2-20

- Final phase of compiler is ~~the~~ the code generator.
- Takes Intermediate code as an input and generate semantically equivalent target program
- Compilers that need to generate efficient target programs, include an optimization phase before code generation
- An optimizer optimizes the intermediate code, from which more efficient code can be generated.



Tasks of a code generator:

- Instruction Selection
- register allocation and assignment
- instruction ordering

- Instruction selection involves choosing appropriate target-machine instructions to implement the intermediate code statements.
- Register Allocation and assignment involves deciding what values to keep in which registers.
- Instruction ordering involves deciding in what order to schedule the execution of instructions
- Code generators partitions ~~instruction~~ ~~to~~ intermediate code into "basic blocks", which consist of sequences of instructions that are always executed together.

Input to Code Generator:

- Intermediate representation of the source program
- Choices of intermediate code representation includes three address code (quadruples, triples, indirect triples, virtual machine representations

(bytecodes or stack-machine code), linear-representation (postfix notation) or graphical representations (syntax trees or DAG's).

1. Instruction Selection: Code generator must map the intermediate code into a code sequence that can be executed by targeted machine. These are the factors for this mapping.
   - level of intermediate code (high level or low level) — need more care / more efficient code can be generated
   - nature of instruction set architecture (data type support)
   - desired quality of generated code (speed or size)

2. Register Allocation:
   - Registers are the fastest computational unit on the target machine (but registers are few)
   - Register allocation is what values to hold in which register.
   - The use of registers is
     Register allocation: choosing set of variables that will reside in registers at each point of time.
     Register assignment: during which we select one register that a variable will reside in.
     Mathematically, assignment is NP-complete problem

3. Evaluation Order:
   - Order in which the instructions are executed can affect the efficiency of target code.
   - Picking a best order is a difficult NP-complete problem since some instructions require fewer registers than other.

Simple Target Machine:

Load :- LD dst, addr

Store : ST x (location), r (register locations)

Computation: OP dst, src_1, src_2

OP → ADD, SUB, MUL, DIV

Unconditional jumps: BR L

L — Branch instruction to label

Conditional jumps: Bcond r, L — condition checking with register r

e.g BLTZ r, L

Program & Instructions Cost:

- Determining the actual cost of compiling & running a program is a complex problem.
- NP-hard (many of the subproblems associated with it).
- For simplicity, cost of one instruction to be one plus the costs associated with addressing mode of operands.

  cost of an instruction = 1 + cost of operands.
  cost of register operand = 0
  cost ~~of~~ involving memory & constants = 1
  cost of a program = Sum of instruction costs.

Examples:

$\underset{1}{LD}$ R0, R1 { 1 + 0 = 1; since register to register no add. cost }

$\underset{1}{LD}$ R0, $\underset{1}{M}$ { 1+1 = 2; since loading from memory },

$\underset{1}{LD}$ R1, $\underset{1}{*}\underset{1}{100}$(R2) { 1+1+1 = 3, contents(contents(100+ contents(R2))) }

Basic Blocks and Flowgraph.

- A graph representation of intermediate code.
- Partition the intermediate code into basic blocks which are the maximal sequences of three address instructions (consecutive).
- Basic blocks becomes nodes of the flow graph.
- Edges indicate which block will follow which other block.
- How to partition intermediate code into basic blocks?

① Determine leader instructions in ~~basic~~ the intermediate code. leader in the first instruction in any basic block.

Date______ 20___

Rules for finding a leader.

(i) First three-address code instruction is a leader

(ii) Any ~~condition~~ instruction that is the target of a conditional or an unconditional jump. Is a leader

(iii) Any instruction that ~~is the target of a condition~~ immediately follows a conditional or unconditional jump is a leader.

Example: Intermediate code:

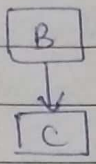
```
1)  i = 1                          leader by rule (i)
2)  j = 1                          leader by rule (ii)
3)  t1 = 10 * i        10, 20      leader by rule (ii)
4)  t2 = t1 + j        11, 12 96
5)  t3 = 8 * t2        88
6)  t4 = t3 - 88       0
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10  goto (3)
10) i = i + 1                      leader by rule (iii)
11) if i <= 10  goto (2).
12) i = 1                          leader by rule (iii)
13) t5 = i - 1                     leader by relue (ii)
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13).
```

Instruction 1, 2, 3, 10, 12, 13 are leaders.
A basic block will contain its leader & all instructions till next leader.

Date ______ 20

Flow graphs:

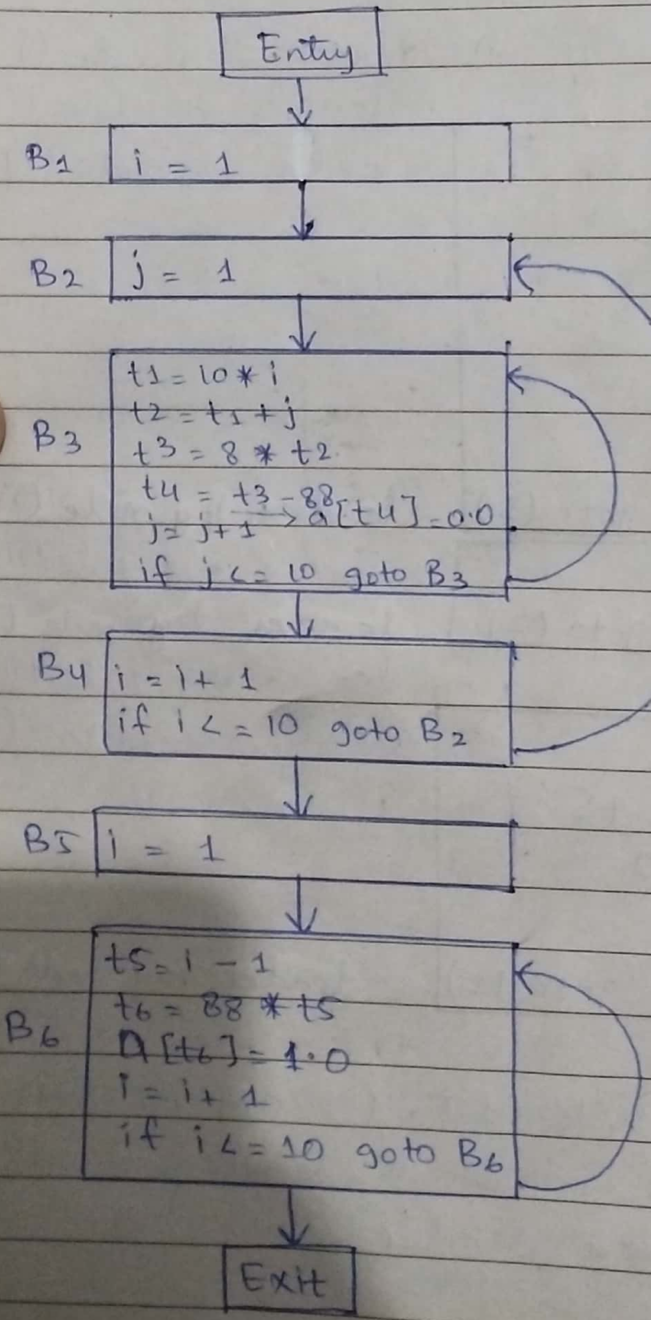
- Once an intermediate code is partitioned into basic blocks, we represent the flow of control between them by a flow graph.
- An edge going from block B to Block C.



① There is a conditional / unconditional jump from Block B (end) to block C (start)

② C immediately follows B in the original order

- B is a predecessor of C or C is successor of B.



```
B1:
i = 1

B2:
j = 1

B3:
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
a[t4] = 0.0
j = j + 1
if j <= 10 goto B3

B4:
i = i + 1
if i <= 10 goto B2

B5:
i = 1

B6:
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B6
```

Exercise 8.4.1.

```
Code: 1. for (i=0; i<n; i++)
      2.     for (j=0; j<n; j++)
      3.         c[i][j] = 0.0;
      4. for (i=0; i<n; i++)
      5.     for (j=0; j<n; j++)
      6.         for (k=0; k<n; k++)
      7.             c[i][j] = c[i][j] + a[i][j] * b[k][j];
```

A matrix multiplication algorithm

Three address code.   n=4

```
1) i = 1
2) j = 1      -> p0 = n * 8    32
                 t0 = p0 + 8   40
3) t1 = n * i
4) t2 = t1 + j                      line 1, 2, 3
5) t3 = 8 * t2.
6) t4 = t3 - ~~88~~ t0
7) c[t4] = 0.0.
8) j = j + 1
9) if j <= n goto (3)
10) i = i + 1
11) if i <= n goto (2)
12) i = 1
13) j = 1
14) k = 1
15) t1 = n * i        4
16) t2 = t1 + j       5
17) t3 = 8 * t2       40
18) t4 = t3 - t0      0
19) t5 = k - 1        ~~0~~ 1
20) t8 = p0 * t5      ~~0~~ 32
21) t6 = t8 + t4      ~~0~~ 32
22) t7 = a[t4] * b[t6]
              0        0
23) c[t4] = c[t4] + t7
24) k = k + 1
25) if k <= n goto (19)
26) j = j + 1
27) if j <= n goto (15)
28) i = i + 1
29) if i <= n goto (13)
        -> ~~line 4, 5, 6, 7~~
```

c

0-7	8-15	16-23	24-31
32-39			

Date ______ 20 ___

Register Allocation & Assignment

- Deciding what values to keep in which registe
- For each three ~~instr~~ address instruction, we'll keep track of what values to keep in what registers.
- An algorithm for a single basic block that will perform above task & avoids unecessary loads & stores.

Principal uses of Registers:

1. In most architectures, almost operands should be in registers to perform operations.
2. Registers make good temporaries, for subresults of longer expressions or to hold a variable used only within a single block.
3. To hold global values, generated in one block & used in other blocks.
4. Help with runtime storage management.

- For each instruction, there is only one machine instruction

→ LD reg, mem

→ ST mem, reg.

→ OP reg, reg, reg.

Register & Address Descriptors:

- We need a data structure, that tells us what program variables have their values in in a register & which registers are then.
- The desired data structure have two descriptors

1. Register descriptor: For each register, it keeps track of the variable name whose current value is in that register.

Initially, all register descriptors are empty. As as generation proceed, each descriptor will hold

the value of zero or more names.
2. Address descriptor: For each variable in program, it keep track of location or locations ~~of~~ where the current value of that variable.

Code Generation.

- getReg(I) : selects registers for each memory location associated with three address instruction 'I'.
- It has access to the register & address descriptors. for all variables of basic block.
- For copy instruction => $x = y$, getReg(I) function will always choose the same register
- For operations instruction => $x = y + z$, getReg(I) will select its registers & code will be Add R_x, R_y, R_z , the value of y must be in R_y & same for z.
- When the basic block is ending, the variables which may be temporary & for that block only, we can free registers for them
- ~~A~~ But what if variable is to be used anywhere e.g. in other block, then we have to store the register value in the memory location.

Managing Register & Address Descriptors:

① For LD R, x.
- Update Register decriptor for R to hold x.
- Update Address descriptor for x add register R as an additional location.

② For ST x, R.
- Update address desciptor for x to include its own memory location

③ For an operation e.g. ADD R_x, R_y, R_z. $x = y + z$
- Update reg. desc. for R_x to hold x.

Date ______ 20 ___

- Update addr. desc. for x so that its only location is Rx. (Note that the memory location for x is not now in the address descriptor for x)
- ~~Update~~ Remove Rx from address descriptor of any variable other than x.

(iii) For copy statement e.g. x = y, after generating load for y into reg. Ry, if needed, & after managing descriptors as for all load statements
- add x to the reg. desc. for Ry.
- update addr. desc. for x so that its only location is Ry.

Example:

Three Address	Assembley	R1	R2	R3		a	b	c	d	t	u	v
t = a - b	LD R1, a	a				a R1	b	c	d			
	LD R2, b	a	b			a R1	b R2					
	SUB R2, R1, R2	a	t			a R1	b			R2		
u = a - c	LD R3, c	a	t	c		a R1	b	c, R3		R2		
	SUB R1, R1, R3	u	t	c		a	b	c, R3		R2	R1	
v=t+u	ADD R3, R2, R1	u	t	v		a	b	c		R2	R1	R3
a=d	LD R2, d	u	a,d	v		R2	b	c	d, R2		R1	R3
d=v+u	ADD R1, R3, R1	d	a	v		R2	b	c	R1		R1	R3
exit	ST a, R2					R2 a	b	c	R1		R1	R3
	ST d, R1					R2 a	b	c	R1 d		R1	R3

Exercise: $x = a + b * c$

Three address code

$t = b * c$

$x = a + t$

Assembley code

$t = b * c$

LD R_1, b

LD R_2, c

MUL R_2, R_1, R_2

$x = a + t$

LD R_3, a

ADD R_3, R_3, R_2

Code	R_1	R_2	R_3	x	a	b	c	t
~~LD R_1, b~~					a	b	c	
LD R_1, b	b				a	b_{R_1}	c	
LD R_2, c	b	c			a	b_{R_1}	c_{R_2}	
MUL R_2, R_1, R_2	b	t			a	b R_2	c	R_2
LD R_3, a	b	t	a		a_{R_3}	b R_2	c	R_2
ADD R_3, R_3, R_2	b	t	x	R_3	a	b R_2	c	R_2

$\textcircled{2}$ $x = a / (b+c) - d * (e + f)$

Three address code:

$t = b + c$

$r = a / t$

$s = e + f$

$u = d * s$

$x = r - u$

Assembley Code:

Date ______ 20__

Code	Reg Desc			Addr. Desc										
	R_1	R_2	R_3	a	b	c	d	e	f	r	s	t	u	x
				a	b	c	d	e	f					
LD R_1, b	b			a	$^{b}R_1$	c	d	e	f					
LD R_2, c	b	c		a	$^{b}R_1$	$^{c}R_2$	d	e	f					
ADD R_2, R_1, R_2	b	t		a	$^{b}R_1$	c	d	e	f			R_2		
LD R_3, a	b	t	a	$^{a}R_3$	$^{b}R_1$	c	d	e	f			R_2		
DIV R_2, R_3, R_2	b	r	a	$^{a}R_3$	$^{b}R_1$	c	d	e	f	R_2				
LD R_1, e	e	r	a	$^{a}R_3$	b	c	d	$^{e}R_1$	f	R_2				
LD R_3, f	e	r	f	a	b	c	d	$^{e}R_1$	$^{f}R_3$	R_2				
ADD R_1, R_1+R_3	s	r	f	a	b	c	d	e	$^{f}R_3$	R_2	R_1			
LD R_3, d	s	r	d	a	b	c	$^{d}R_3$	e	f	R_2	R_1			
MUL R_1, R_3, R_1	u	r	d	a	b	c	$^{d}R_3$	e	f	R_2			R_1	
SUB R_3, R_2, R_1	u	r	x	a	b	c	d	e	f	R_2			R_1	R_3