General issues

- **Target language:**
  - absolute machine language
    - all addresses refer to actual addresses
    - program placed in a fixed location in memory
  - relocatable machine language (object modules)
    - sub-programs can be compiled separately, libraries can be used
    - linking/loading necessary, but much greater flexibility
  - assembly language
    - code is easy to generate/read
    - additional pass required (assembler)

- **Instruction selection:** depends on
  - uniformity of instruction set
  - availability of special instructions, e.g.,
    - INC a vs MOV a R0 ADD#1 R0 MOV R0 a
General issues

- Register allocation
  - register allocation: deciding which variables are stored in registers
  - register assignment: assigning specific registers to variables

Example: Integer division on IBM Sys/370: \( \text{DIV } x \ y \)
- \( x \) - even register of an even/odd register pair that holds 64 bit dividend
- \( y \) - divisor
- after division, \( x \) holds remainder, corresponding odd register holds quotient

\[
\begin{align*}
\text{LOAD } R0 & \ a \\
t = a / b & \Rightarrow \ SRDA \ R0 \ 32 \\
\text{DIV } R0 & \ b \\
\text{ST } R1 & \ t
\end{align*}
\]
Target machine

- Byte-addressable, 4 bytes / word
- \( n \) general purpose registers
- Instruction format: OP SRC DEST
- Addressing modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Syntax</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>register</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>indexed</td>
<td>c(R)</td>
<td>c + contents(R)</td>
</tr>
<tr>
<td>register indirect</td>
<td>*R</td>
<td>contents(R)</td>
</tr>
<tr>
<td>indexed indirect</td>
<td>*c(R)</td>
<td>contents(c + contents(R))</td>
</tr>
<tr>
<td>constant/literal</td>
<td>#c</td>
<td>constant c</td>
</tr>
</tbody>
</table>

Examples:
MOV 4(R0) M    MOV *4(R0) M    MOV #1 R0
Basic blocks

**Definition:** sequence of consecutive statements such that flow of control enters at the beginning and leaves at the end without halt or possibility of branch except at the end

**Leader:** first statement of a B.B.

**Determining basic blocks:**

1. Determine leaders:
   (i) first statement is a leader
   (ii) targets of conditional/unconditional branch
   (iii) any statement immediately following a branch

2. For each leader, all statements following it upto (but not including) next leader or end of program constitutes a basic block.
Flow graphs

Definition: directed graph with

1. a node corresponding to each basic block, with one node distinguished as *initial*

2. an edge from $B_1$ to $B_2$ if
   (i) there is a jump from last statement in $B_1$ to first statement in $B_2$, or
   (ii) $B_2$ immediately follows $B_1$ in program text, and $B_1$ does not end in an unconditional jump
**Next-use information**

**Def.** A statement \( x = y + z \) is said to **define** \( x \) and **use** or **reference** \( y \) and \( z \).

**Live variable:** A variable is live at a given point if its value is used after that point in the program.

**Algorithm:**
1. Scan each B.B. backward from last statement to the first
2. For each stmt \( i: x = y \ OP \ z \) in the backward pass
   (i) attach to stmt \( i \) the information currently found in the Symbol Table for \( x, y, z \)
   (ii) in the ST, set \( x \) to **NOT LIVE**
   (iii) set \( y, z \) to **NEXT USE = i**

**Applications:** (i) storage for temporaries  (ii) code generation
Storage for temporaries

Principle: pack two temps into same location if they are not simultaneously live

Assumption: temps are defined and used within basic blocks

Method:
for each temporary variable
assign it to first location that does not contain a temp.
(create new location if needed)

Example: $x = a*a + 2*a*b + b*b$
Intermediate language

Assignment statements
\[ x = y \text{ op } z \quad x = \text{ op } y \quad x = y \]

Array references
\[ x = y[i] \quad x[i] = y \]

Pointer operations
\[ x = &y \quad x = *y \quad *x = y \]

Jumps
\[ \text{goto } L \quad \text{if } x \text{ relop } y \text{ goto } L \]

Procedure calls
\[ \text{param } x_1 \]
\[ \text{param } x_2 \]
\[ \vdots \]
\[ \text{param } x_n \]
\[ \text{call } p, n \]
Assignment statements

Input: sequence of 3-addr statements constituting a basic block
Assumptions: for each operator used in 3-addr stmt, there is an equivalent target language operator

Auxiliary information:
- Register descriptors (RD):
  - shows which variables are stored in each register
  - initially, all registers are empty
- Address descriptors (AD):
  - for each name, shows the location(s) where the current value of the name is stored (register/memory/stack etc.)
  - can be stored in symbol table

Auxiliary function: getreg() - given a 3-addr statement, determines a location $L$ where the result of the 3-addr statement should be stored
Assignment statements

Step I: \( x = y \ op \ z \)

1. Let \( L = \text{getreg}() \).

2. Let \( y' = \text{location}(y) \) (preferably register). If \( y' \neq L \), generate

\[
\text{MOV } y' \ L
\]

3. Let \( z' = \text{location}(z) \) (as above). Generate

\[
\text{OP } z' \ L
\]

4. Update address descriptor of \( x \) to \( \{L\} \); remove \( x \) from all RDs.

5. If \( L \) is a register, update its RD.

6. If \( y \) (or \( z \)) is
   (i) in a register
   (ii) has no next use and is not live on exit from the block
   change RD to indicate that the register no longer contains \( y \)
   (or \( z \)).
Assignment statements

Step I (special case): \( x = y \)

1. If \( y \) is in register \( R_i \):
   (i) change RDs and AD for \( x \) to indicate that \( x \) is now only in \( R_i \);
   (ii) if \( y \) has no next use and is not live on exit from block, delete \( y \) from RD for \( R_i \).

2. If \( y \) is in memory:
   (i) load \( y \) into a register (obtained using getreg()), and proceed as above; OR
   (ii) generate \( \text{MOV } y \ x \) (preferable if \( x \) has no next use in the block).
Assignment statements

Step II: after processing all stmts in the basic block, generate MOV instructions to store all variables that are live on exit, but not currently in their memory locations.

for each variable x in each register
  check AD for x to determine whether its current value is in memory
  if not, generate suitable MOV instruction
1. If \( y \) is in a register \( R \) and
   - \( R \) holds no other names
   - \( y \) is not live / no next use after this statement
then
   (i) delete \( R \) from AD for \( y \);    (ii) return \( R \).

2. If there is an empty register, return it.

3. If \( x \) has a next use, or \( \text{op} \) is an operator (e.g. indexing) that requires a register:
   (i) find an occupied register \( R \);
   (ii) if value(s) in \( R \) are not also in memory, generate
       \[ \text{MOV } R, M \]
   (iii) update AD for \( M \);    (iv) return \( R \).

4. If \( x \) is not used in the block, or no suitable occupied register can be found in step 3, return memory location of \( x \).
### Arrays

<table>
<thead>
<tr>
<th>INSTR</th>
<th>i in $R_i$</th>
<th>i in $M_i$</th>
<th>i on stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b[i]$</td>
<td>MOV b(Ri) R</td>
<td>MOV Mi R</td>
<td>MOV Si(A) R</td>
</tr>
<tr>
<td></td>
<td>MOV b(R) R</td>
<td>MOV b(R) R</td>
<td></td>
</tr>
<tr>
<td>$a[i] = b$</td>
<td>MOV b a(Ri)</td>
<td>MOV Mi R</td>
<td>MOV Si(A) R</td>
</tr>
<tr>
<td></td>
<td>MOV b a(R)</td>
<td>MOV b a(R)</td>
<td></td>
</tr>
</tbody>
</table>

A - register containing pointer to AR for $i$
Si - offset of $i$ within AR
R - location returned by getreg()
### Pointers

<table>
<thead>
<tr>
<th>INSTR</th>
<th>p in $R_p$</th>
<th>p in $M_p$</th>
<th>p on stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = *p$</td>
<td>MOV *Rp a</td>
<td>MOV Mp R</td>
<td>MOV Sp(A) R</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOV *R R</td>
<td>MOV *R R</td>
</tr>
<tr>
<td>$*p = a$</td>
<td>MOV a *Rp</td>
<td>MOV Mp R</td>
<td>MOV a R</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOV a *R</td>
<td>MOV R *Sp(A)</td>
</tr>
</tbody>
</table>

A - register containing pointer to AR for $p$

Sp - offset of $p$ within AR

R - location returned by getreg()
Conditional jumps

Assumptions:

1. **CCR** (Condition Code Register) indicates whether the last quantity computed or loaded into a register is less than, greater than, or equal to 0.

2. Compare instruction: \( \text{CMP} \ x \ y \)
   - sets CC to +ve if \( x > y \), etc.

3. Conditional jump instructions:
   \[
   \begin{align*}
   &\text{JLT} \ L & \text{JLE} \ L & \text{JEQ} \ L & \text{JGE} \ L & \ldots
   \end{align*}
   \]

Translation: \( \text{if } x \ \text{op} \ y \ \text{goto} \ L \)

\[
\begin{align*}
\text{CMP} & \ x \ y \\
\text{J<op>} & \ L
\end{align*}
\]
Procedure calls

Scheme:
- Position of AR for current procedure is stored in SP
- SP points to beginning of AR on top of stack
- Use positive offsets from SP to access fields of AR
- Calling procedure increments SP and transfers control
- On return, caller decrements SP

[Alt. SP points to top of stack]
Procedure calls

Scheme:
- Position of AR for current procedure is stored in SP
- SP points to beginning of AR on top of stack
- Use positive offsets from SP to access fields of AR
- Calling procedure increments SP and transfers control
- On return, caller decrements SP

[ Alt. SP points to top of stack ]

Initialization:
MOV #stackstart SP
/* code for main */
.
.
.
HALT
Procedure calls

Caller
ADD #caller.recordsize SP
MOV R0 4(SP)  /* 1st argument */
MOV R1 8(SP)  /* 2nd argument */
MOV #here+16 *SP  /* return address */
GOTO <addr. of 1st statement of callee>
SUB #caller.recordsize SP

Callee
/* save all registers */
/* do required work */
MOV R0 4(SP)  /* return value */
/* restore registers */
GOTO *0(SP)  /* return statement */