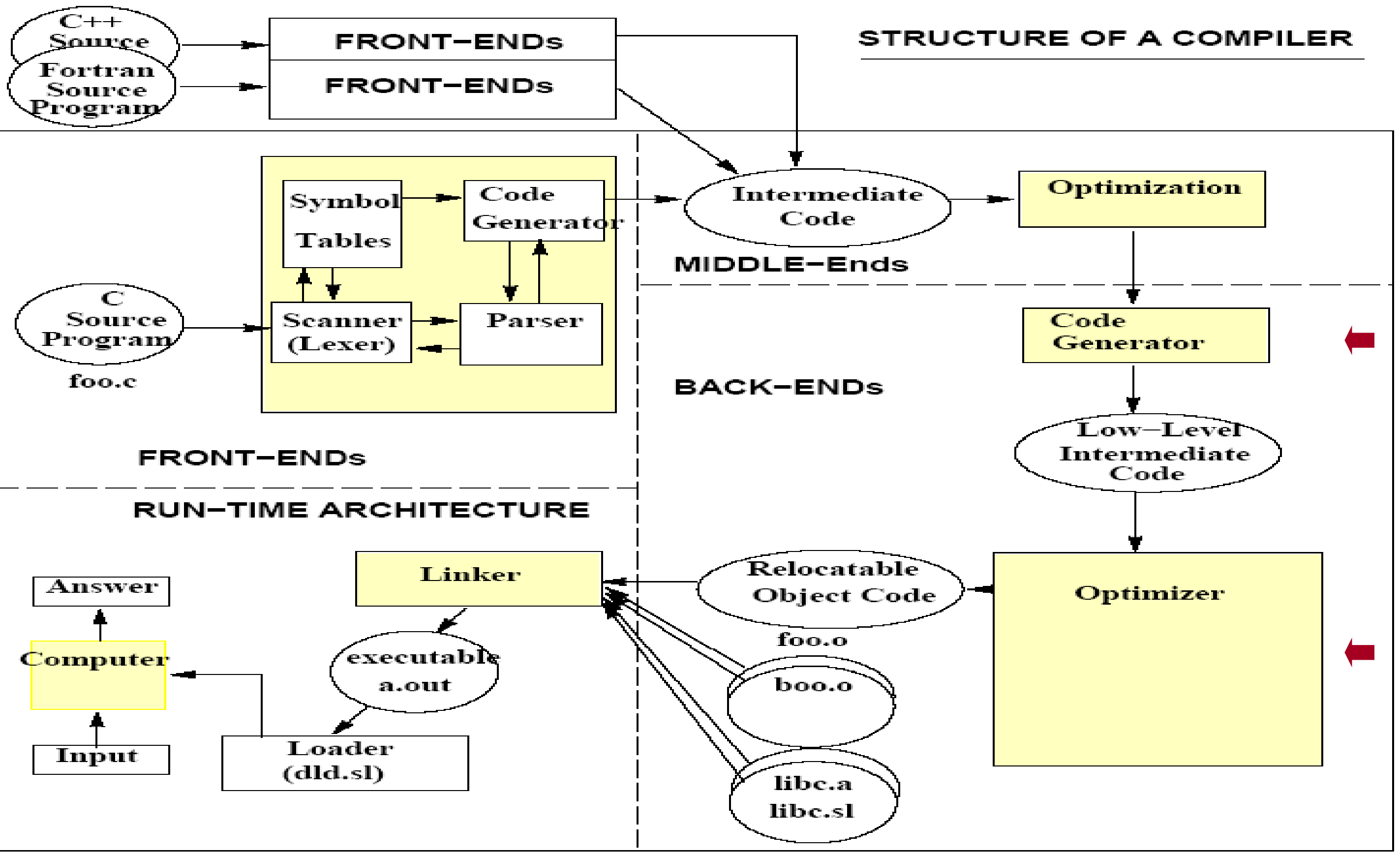
Overview of Compiler Optimization

- Code Generation and Optimization
- An Example of Code Optimization
- Overview of Optimization Concepts

Machine Code Generation & Optimization

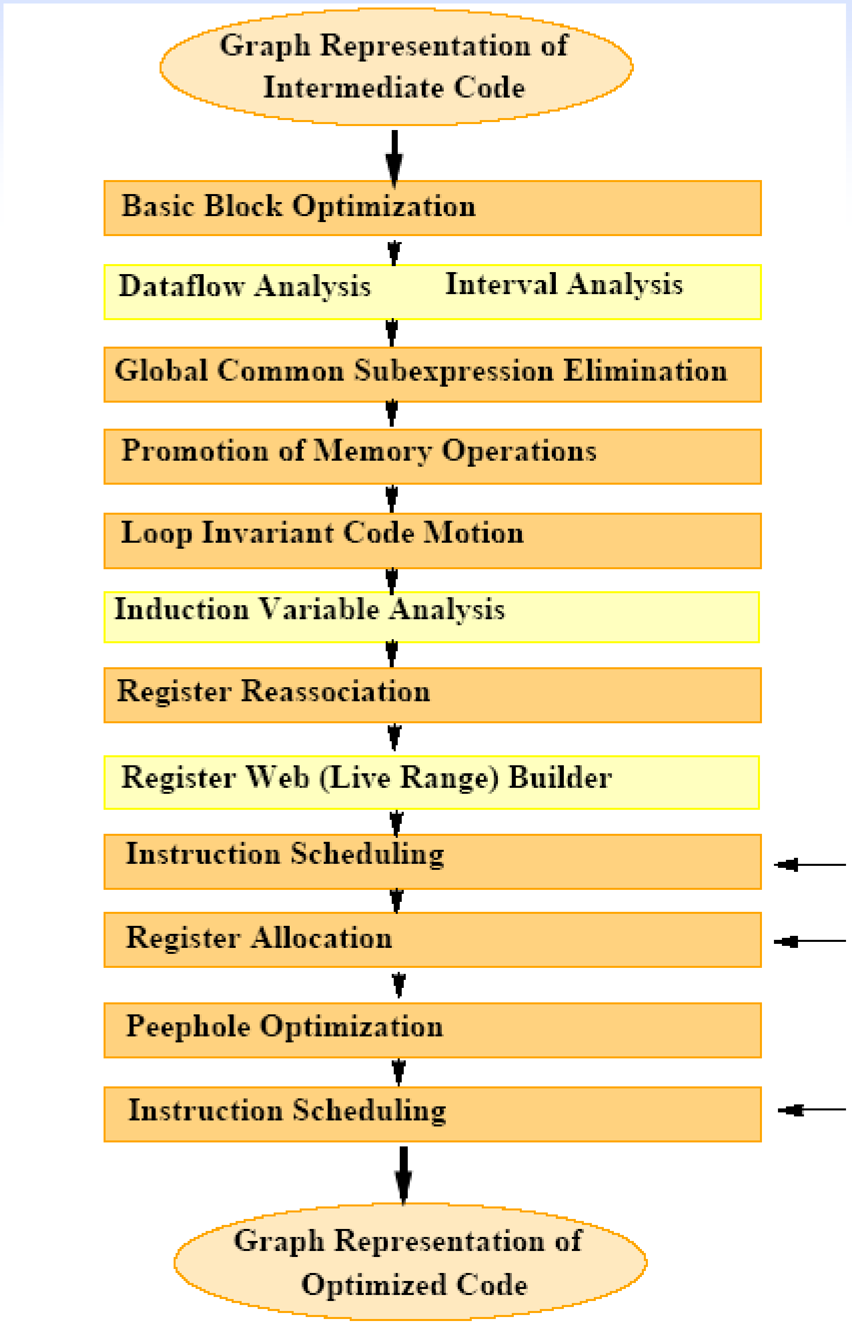
- Intermediate representation (IR) such as stack machine code is translated into machine code
  - It is still pseudo machine code where registers are not yet allocated (we can call it a low-level IR)
- Pseudo machine code is optimized and transformed into real machine code

Structure of Modern Compilers



Optimization Phases

- Un-optimized pseudo machine code becomes better code by passing thru optimization phases
  - What kind of optimization phases we have and how to order those phases differ from compiler to compiler
  - We will show one example of optimization phases with an example code optimization


Graph Representation of Intermediate Code
Basic Block Optimization
Dataflow Analysis
Interval Analysis
Global Common Subexpression Elimination
Promotion of Memory Operations
Loop Invariant Code Motion
Induction Variable Analysis
Register Reassociation
Register Web (Live Range) Builder
Instruction Scheduling
Register Allocation
Peephole Optimization
Instruction Scheduling
Graph Representation of Optimized Code

One Pitfall of Optimizations

- There are numerous optimizations that do not seem to arise in practice if you "program very well", such as common subexpression elimination (CSE), dead code elimination and copy propagation, constant propagation, constant folding, etc.
  - CSE: x=y+z; … w=y+z -> x=y+z; …w=x;
  - Copy propagation: x=y;… z=x+100 -> x=y;… z=y+100
  - Constant folding: if (4>3) -> if (true)
- The reality is that although you may be able to avoid explicit ones while you do your programming, the compiler still generate those opportunities (e.g., address computation)
- We will see such examples soon

An Example of Code Optimization

```
--------------- Source C Code -----------------
int a[25][25];
main()
{
      int i;
      for (i=0; i<25; i++)
            a[i][0] = 0;
}

------------- Optimized Assembly Code ----------------

            ADDIL   LR'a-$global$,%r27              ;offset 0x0
            LD0     RR'a-$global$(%r1),%r31         ;offset 0x4
            LDI     -25,%r23                        ;offset 0x8
$00000003
            ADDIB,<         1,%r23,$00000003        ;offset 0xc
            STWM    %r0,100(%r31)                   ;offset 0x10
```

Un-optimized Code

```
          STW             0,-40(30)
          LDW             -40(30),206
          LDI             25,212
          IFNOT           206 < 212 GOTO $00000002
          NOP
$00000003
          LDW             -40(30),206
          ADDILG          LR'a-$global$,27,213
          LDO             RR'a-$global$(213),208
          MULTI           100,206,214
          ADD             208,214,215
          STWS            0,0(215)
          LDW             -40(30),206
          LDO             1(206),216
          STW             216,-40(30)
          LDW             -40(30),206
          LDI             25,212
          IF              206 < 212 GOTO $00000003
          NOP
$00000002
```

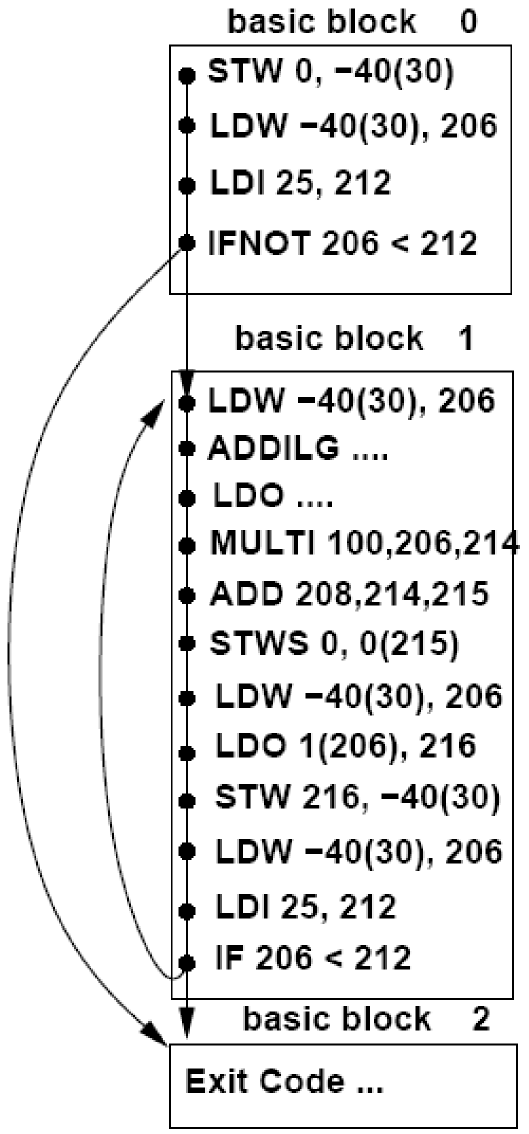
Representation: a Basic Block

- *Basic Block* = A Consecutive Sequence of Instructions (Statements)
  - A sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end
  - A *basic block header*: target instruction of a branch or a control join point

Optimizations within a basic block are *local* optimizations.

- How to build basic blocks?
  - First build a control flow graph of instructions, then identify basic block headers
  - Many optimizations work on a control flow graph of basic blocks

Building Basic Blocks for the Example Code



Local Optimizations

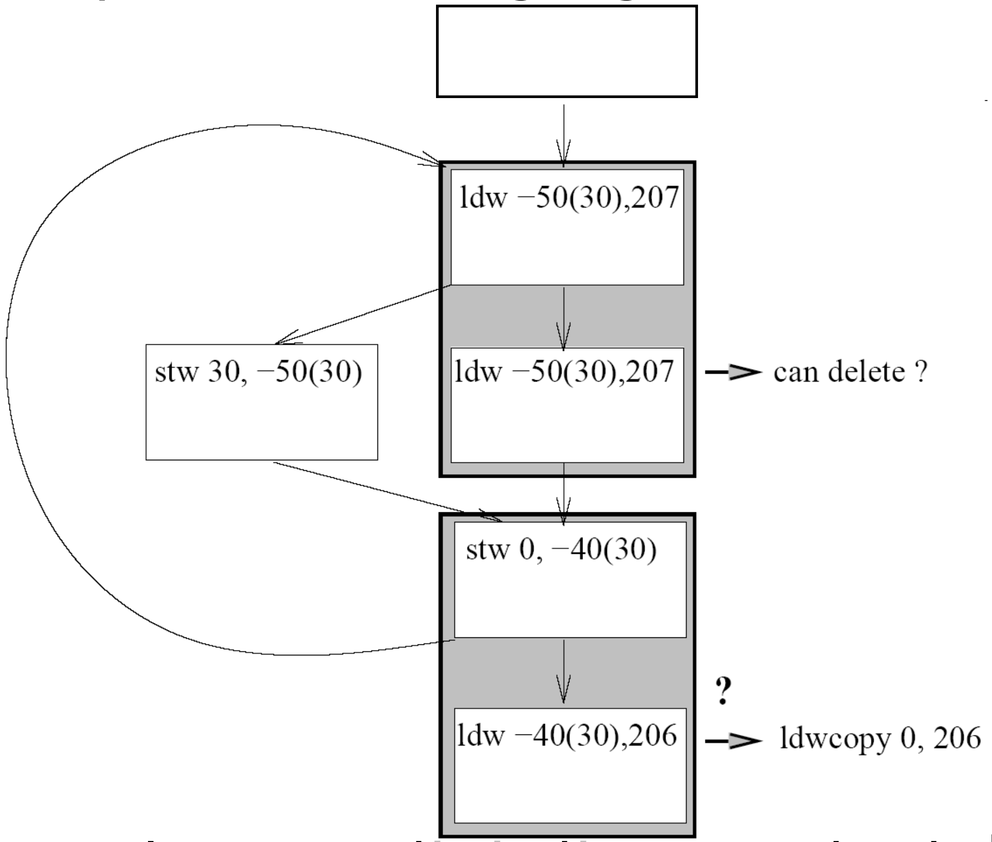
- Analysis and transformation performed within a basic block
- No control flow information is considered
- Examples of local optimization
  - Load to copy optimization
  - Local common sub-expression elimination
    - Some expressions evaluated more than once in a BB is replaced by a single calculation (delete later ones if they have the same target register)
  - Local constant folding or elimination
    - Expressions that can be evaluated at compile-time is replaced by constant, compile-time value
  - Dead code elimination

After Local Optimizations

```
          STW               0,-40(30)
          LDWCOPY           0,206                 :: <==     LDW        -40(30),206
          L야               25,212                           load-copy optization
$00000003
          LDW               -40(30),206
          ADDILG            LR'a-$global$,27,213
          LD0               RR'a-$global$(213),208
          MULTI             100,206,214
          ADD               208,214,215
          STWS              0,0(215)              :: Deleted LDW        -40(30),206
          LD0               1(206),216                       local CSE optimization
          STW               216,-40(30)
          LDWCOPY           216,206               :: <==     LDW        -40(30),206
          LDI               25,212                           load-copy optimization
          IF 206 < 212 GOTO $00000003
          NOP
$00000002
```

Extended Basic Block

- A chain of sequential basic blocks that has no incoming branches yet can have outgoing branches



- Can we apply same optimizations on extended basic blocks? Yes

Global CSE

Redundant Definition Elimination

- Value numbering:
  - Hints for applying CSE: code generator is expected to assign the same target pseudo register for the same right-hand-sides
- Compute available expressions across all paths an expression x+y is available at a point *p* if every path to *p* evaluates x+y and after last evaluation prior to reaching *p*, there are no subsequent assignment to x or y
- Delete redundant expressions

After Global CSE Optimization

```
    STW         0,-40(30)        ; @i
    LDWCOPY  0,206               ; @i
    LDI         25,212
$00000003
    ADDILG      LR'a-$global$,27,213  :: deleted LDW  -40(30),206
    LDO         RR'a-$global$(213),208
    MULTI       100,206,214
    ADD         208,214,215
    STWS        0,0(215)         ; @a[i][0]
    LDO         1(206),216
    STW         216,-40(30)      ; @i
    LDWCOPY  216,206             ; @i
    IF 206 < 212 GOTO $00000003   :: deleted LDI  25,212
    NOP
```

Promotion of Memory Operations

- Memory Live Range: a set of stores (definition) and loads (uses)
  - Access the same location in memory
  - For each use (load) in the set, all definitions that might *reach* it are also in the set
- A Live Range of Memory can be promoted to register operations if certain conditions are met
  - Store is promoted into copies and loads are deleted
  - The front-end provides some information on which loads and stores access the same location, or you can find it by yourself by analyzing the assembly at this phase

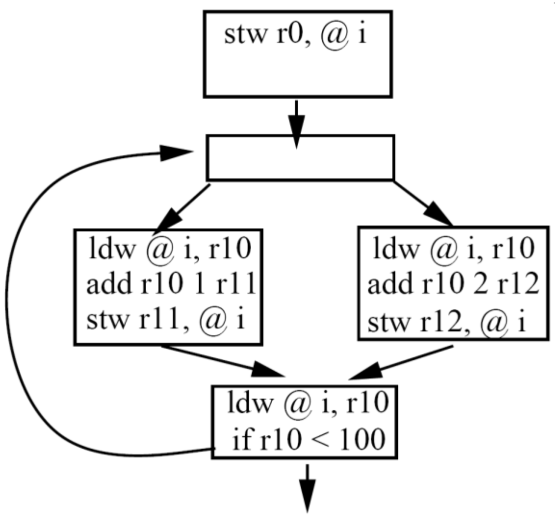
Example

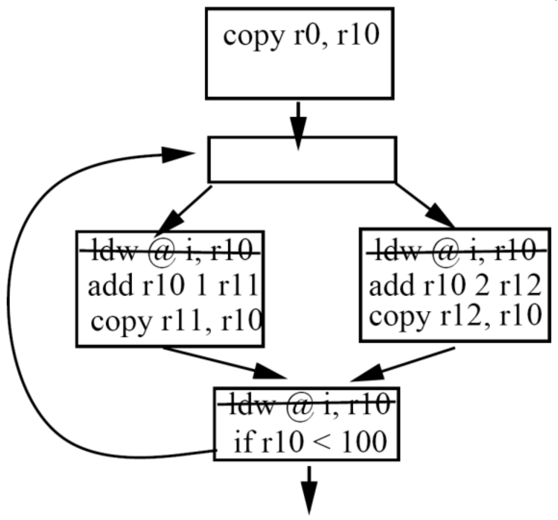
Example

```
int i =0;

Do {

   if (...)
       i = i + 1;
   else
       i = i + 2;
}  while (i < 100)
```





After Register Promotion

- There are two Memory Live Ranges

```
STW       0,-40(30)      ; @i     STW          216,-40(30)     ;@i
LDWCOPY 0,206            ; @i     LDWCOPY      216,206         ;@i
```

- After Optimization

```
      Copy        0,206
      LDI         25,212
      NOP
$00000003
      ADDILG      LR'a-$global$,27,213
      LDO         RR'a-$global$(213),208
      MULTI       100,206,214
      ADD         208,214,215
      STWS        0,0(215)
$00000001
      LDO         1(206),216
      COPY        216,206
      IF   206 < 212 GOTO $00000003
      NOP
```

Loop Transformation

Traditional loop optimizations

- Loop invariant code motion (LICM)
- Strength reduction
- Induction variable elimination

After Loop Optimization

```
          COPY          0,206
          LDI           25,212
          ADDILG        LR'a-$global$,27,213         :: ← loop invariant code
          LDO           RR'a-$global$(213),208       :: ← motion into loop header
          LDI           0,218                        :: ← newly inserted
          LDI           2500,219                     :: ← at the loop header
$00000003
          COPY          218,214                      :: ← MULTI        100,206,214
          ADD           208,214,215
          STWS          0,0(215)

          LDO           100(218),220                 :: ← LDO          1(206),216

          COPY          220,218                      :: ← COPY         216, 206
          IF 218 < 219 GOTO $00000003                :: ← IF 206 < 212 GOTO $00000003
```

Building Register Live Ranges

- Live range of register definition and uses
- Unit for register allocation
- Helps for dead code elimination

After Building Register Live Ranges

- Two Dead Instructions

```
        COPY            0,206
        LDI             25,212
```

- After Optimization

```
        ADDILG          LR'a-$global$,27,66
        LDO             RR'a-$global$(66),65
        LDI             0,69
        LDI             2500,71
$00000003
        COPY            69,67
        ADD             65,67,68
        STWS            0,0(68)
        LDO             100(69),70
        COPY            70,69
        IF 71 < 69 GOTO $00000003
```

Instruction Scheduling

- Assume that the machine has two ALUs and superscalar capability.

```
        ADDILG          LR'a-$global$,27,66
        LDO             RR'a-$global$(66),65
        LDI             0,69
        LDI             2500, 71
$00000003
        COPY            69,67
        LDO             100(69),70      :: ← code motion
        ADD             65,67,68
        COPY            70,69           :: ← code motion
        STWS            0,0(68)
        IF 71 < 69 GOTO $00000003
```

Register Allocation

- Copy Elimination : two ways
  - Copy Propagation : make copy dead
  - Copy Coalescing : if the target live range and the source live range of a copy instruction does not interface, those ranges are merged together and are allocated the same register: copy is deleted since it has the form of "copy r1 r1" now.
- Building interference graphs
- Color live ranges such that two interfering ranges are not assigned by the same color

After Register Allocation

- "COPY 69,67" is propagated and deleted
- "COPY 70,69" is coalesced as "COPY 69 69"

- Result after Copy Elimination

```
        ADDILG          LR'a-$global$,27,66
        LDO             RR'a-$global$(66),65
        LDI             0,69
        LDI             2500,71
$00000003
        ADD             65,69,68
        LDO             100(69),69
        STWS            0,0(68)
        IF 71 < 69 GOTO $00000003
```

Final Finishing

- Register save and restore code generation
  - Call boundary and method entry/exit
- Peephole optimizations
- Assemble