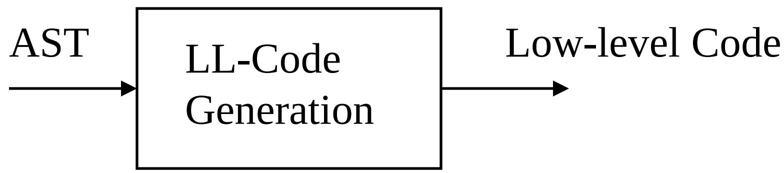
Low-Level Code Generation

- Ok, so the Runtime Environment from the last chapter was our roadmap for how we are going to do Low-Level Code Generation this chapter
  - Our major task now will be to walk the AST and transform it into our low-level language
- After this, we will do Optimization, Register Allocation, Code Scheduling, and Assembly Generation
  - Then we will use an existing assembler and loader/linker to create an executable
- But our first step, if we are going to generate low-level code, is to define its format



Low-Level Code Format

- First, let's talk about what we call this format
  - Some people call it intermediate code
    - High-level, intermediate-level, then assembly
  - Some call it low-level code
    - Close to assembly in form
  - Some might call it back-end code
  - Some call it RTL (register transfer language)

- Probably the differences in terminology vary because the actual format varies
  - Some use a tree structure
    - Might be more inclined to call it an intermediate form
  - Others use a format very close to assembly
    - Might call it low-level or RTL

Low-Level Code Format

- I will probably say mostly "low-level"
  - Or not ….

- Formats for low-level code
  - A tree structure
    - Used by GCC and some compiler texts
    - Looks a lot like AST, except nodes correspond more to assembly-level constructs
      - e.g., no for-loops (jmps instead)
    - Can be good for CSE (common subexpression elimination)
    - Once you have this tree built and set all its attributes, you can generate assembly code with a simple tree walk
    - May not be as easy to do some optimizations

Low-Level Code Format

- Formats for low-level code (cont)
  - Operation-based format similar to assembly
    - List of operations, each of which corresponds to an assembly-like instruction
    - May facilitate optimization
      - Better visibility for things like code scheduling
    - Assembly generation from this format is a very simple translation process
      - May be done in multiple steps, making the low-level code more and more architecture-specific
- Which is better?
  - Define "better"
    - If looking for highly optimized code, probably better to use an operation-based format
    - If looking for fast compilation, maybe tree-based is better
  - I like operation-based – Project #3 should be done this way

Low-Level Code Format

- Text presents 2 formats for low-level code
  - Three-address code
    - In a classic RTL format
    - They leave variable names in the format, rather than referring to them by their register or memory location
    - Similar to what I would like to see in project
      - But use reg #s rather than var names
    - Examples: 2*a + (b-3) and repeat {x := x –1} until x=0;

```
t1 = 2 * a
t2 = b – 3
t3 = t1 + t2
```

```
label L1
t1 = x – 1
x = t1
t2 = (x == 0)
if_false t2 goto L1
```

Low-Level Code Format

- Text presents 2 formats (cont)
  - P-Code
    - Author should be shot for putting it in the text
    - It is a low-level format designed for a stack architecture
      - i.e., where all computation is done stack-based, like an HP calculator
      - No one builds these much anymore
        - Maybe a special-purpose processor, but not likely
      - To do c = a + b, do:

        push a, push b, add, pop c
    - One example, and then we will forget we ever saw this (except maybe exam)

      2*a + (b-3)

```
ldc 2
lod a
mul
lod b
ldc 3
sbi
adi
```

Low-Level Code Format

- Let's design our own (based on Lcode, used by IMPACT compiler from Illinois)
  - Need two formats: internal data structure and file format
- Most languages allow two top-level structures
  - Data declarations and function declarations
  - We probably need classes which represent these two things
- Inside of functions, we have a sequence of instructions
  - Need an Operation class
    - Operations are made up of Operands
- Also inside of functions, you may need labels as targets of gotos
  - But instead of using labels, a powerful data structure is to group sets of instructions into blocks called basic blocks
    - We can define a Block class which contains a sequence of Operation objects
    - Jumps/branches simply designate a Block as the target

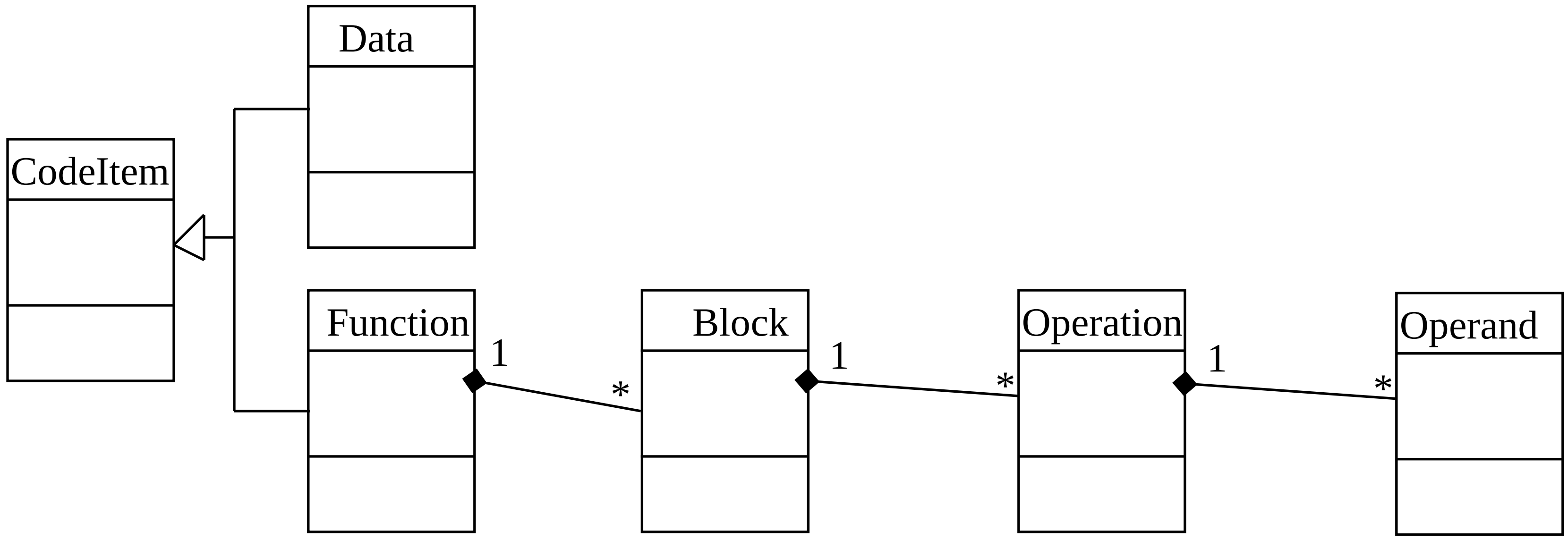
Low-Level Code Format

- Definition of Basic Block
  - A list of sequentially-executed instructions, which ends at a jump, branch, or return, or when the next instruction is the target of a jump or branch
  - Control flow always enters a BB at the top and exits at bottom
    - If you begin executing a BB, you will execute the entire thing

```
BB1:
  add R1, R2, R3
  R4 = load (FP + 4)
  bz BB4

BB2:
  sub R5, R4, R1
  jmp BB5

BB3:
  add R5, R2, R3
  sub R7, R5, R1
  store (SP – 4), R7

BB4:
  pop FP
  return
```

Low-Level Code Format

- Notice that in previous example the destination of control instructions (jumps, branches) was a BB
  - Makes for a very clean way of representing the code
  - Facilitates optimizations which may make big changes to code structure – e.g., changing the polarity of a branch
- Here are the classes we have defined so far



Low-Level Code Format

- A start at classes

```
public class Function
        extends CodeItem{

    private String name;
    private FuncParam param;
    private CodeItem nextItem;
    private Block firstBlock;
    private Block lastBlock;
    private int localVarSize;
    private int spillSize;
    private int frameSize;
    private Attribute attr;

    public Function () { }

      // accessor methods
}
```

```
public class Block  {

    private Function func;
    private Block prevBlock;
    private Block nextBlock;
    private Operation firstOper;
    private Operation lastOper;
    private int blockNum;
    private Attribute attr;

    public Block (int num) {
      blockNum = num;
    }

      // accessor methods
}
```

Low-Level Code Format

- Operation class would need what instance variables?
- Operation number, operation type, pointers to operands, possibly some source-code info like variable name, line number
  - Note: for generality, might want an array of operands, with number unspecified

```
public class Operation {

    private Block block;
    private Operation prevOper;
    private Operation nextOper;
    private int opNum;
    private int opType;
    private Operand dest;
    private Operand src1;
    private Operand src2;
    private Operand src3;     // stores have 3 sources
    private Attribute attr;
        // plus constructors and accessor methods
}
```

Low-Level Code Format

- Operand class gets a bit tricky
  - Registers, special regs (e.g., SP), Blocks, function names, global labels, string constants, int/double constants, etc
  - Won’t go into more now

- That gives you an idea of how the data structure might work
  - During code generation, you walk the AST and begin creating the low-level data structure
    - Tricky part is non-sequential instructions – have to get all the Blocks laid out correctly
    - Much of the rest is just mapping an AST construct into corresponding low-level construct

- Still need to define file format
  - Note: for Project #3, you just need to create the internal structure; I have created the file format

Low-Level Code Format

- Low-level file format
  - Should capture object hierarchy with indents
  - Needs to allow variable number of elements in a field (variable number of operands in an operation)
  - Needs to be simple to parse in
    - Parentheses help not only visually, but can help with optional fields and variable number of elements
      - Can use empty parentheses for optional field
    - Try to keep format as close as possible to internal data structure
  - Needs to be easily readable

Low-Level Code Format

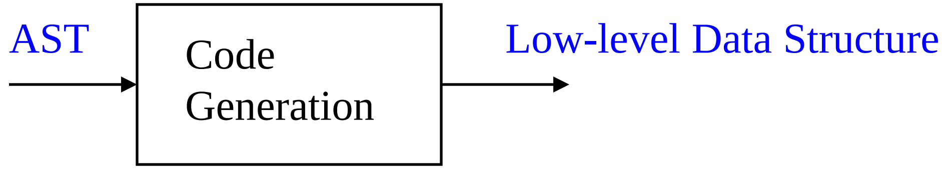
- File format example

```
(data integer ["i"] [0])
(function func1 [(frameSize 0) (regAlloc 0) (return_type (ptr(int))) ]
   (bb 0 [ ]
      (macro funcEntry)
   )
   (bb 1 []
      (oper 1 add_int [(reg 0)]  [(reg 1) (reg 2)] [ ] )
      (oper 2 push  [ ]  [(reg 0)]  [ ] )
      (oper 3 jmp  [ ]  [(bb 2)]  [ ] )
   )
   (bb 2 [ ]
      …
   )
   (bb 3 [ ]
      (macro funcExit)
      (oper 85 return  [ (sreg SP)]  [(sreg SP)]  [ ] )
   )
)
```

Low-Level Code Generation

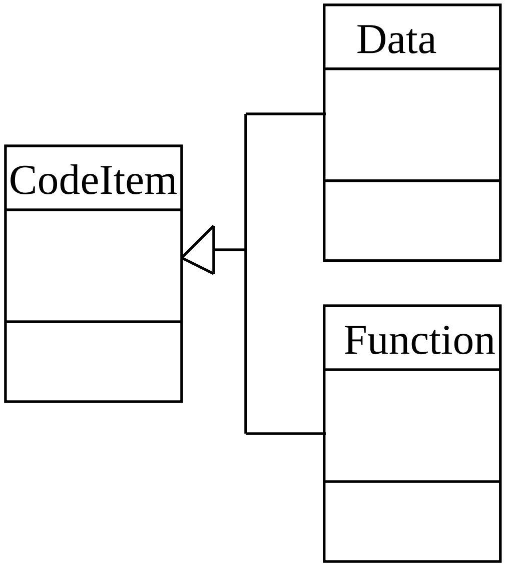
- So, the task of code generation is to walk the AST, creating a new data structure as you go
  - This low-level data structure can then be written to file (academic environment) or passed on to the next phase of the compiler

- OK, so how do we do this?
  - We could make the low-level code an attribute, and generate during parse using an attribute grammar
  - Better to perform separate pass



Low-Level Code Generation

- Look at language definition on pg 492, and think about what we will encounter as we walk the AST
  - First, we will see a declaration_list, either global vars or functions
    - If it's a var_decl, then add a new Data object, linking to other CodeItems in list
    - If it's a fun_decl, create a new Function
      - Next get return type and name from appropriate fields
      - Make FuncParams
      - Create new Block and make it currentBlock
      - Then call genCode on CompoundStmt



Low-Level Code Generation

- So now we're executing genCode inside the CompoundStmt, and we have a pointer to the current basic block (bb 1)
  - A compound_stmt is made up of local_decls and stmts
    - Each of local_decls will define a local variable of the current function
      - Need to start making a plan for how this variable will be handled – mark this as attribute or in symbol table
      - If it is a scalar, plan to put it in register – give it a virtual register # - can't if declared static or if &x ever done
      - If it can't go into register, designate a location in stack frame for it
        - Means we need some global-type variables to keep track of local variable space needed for this function so far

Low-Level Code Generation

- Working on executing genCode inside CompoundStmt (cont)
  - Completed local vars
  - Next work on statements
    - We have a Function, and a currentBlock
    - Now we will start adding Operations to the current Block
    - The nature of the statements (e.g., IF, WHILE) will make it clear when we need to start a new Block
    - Call genCode on Stmts to generate Operations
- genCode for ReturnStmts
  - If it returns an expression, call genCode on the Expr
    - This may generate Operations
    - This will result in the expression being assigned to a register or memory location
  - Add Operation to move expression result into return register
  - Create Exit block if necessary- nah, I already did
  - Add jump Operation to exit block

Low-Level Code Generation

- genCode for ExpressionStmt
  - Just call genCode on the Expr and do nothing else
- Will come back to the control flow statements later
- Think about a few of the Expression types you need to handle
- genCode for BinaryExpression
  - Call genCode on left and right child
  - Get location of where children stored their results
  - Get a new register for your result
  - Add Operation to do your function
- genCode for VarExpr
  - Just look up your location in the symbol table (if not already done in previous pass); if global, create a load oper
- genCode for NumExpr
  - Probably don't have to do much
  - You could assign yourself to a register, or let parent handle

Low-Level Code Generation

- genCode for CallExpr
  - A bit more complicated
  - Call genCode on params to generate code for them
    - In reverse order if X86
  - Add Operation to move each param to register or memory
    - For project, just “Pass R1”
  - Add Call Operation
  - May want to add a Macro Operation for PostCall
    - Or let a later pass just handle this
    - For project, you will annotate Call with param size
  - Need to move return register into regular register
  - What about saving registers, ala caller-save convention?

Low-Level Code Generation

- OK, back to control flow statements
  - A couple of things we haven't seen
    - We need to determine a convention for Boolean decisions
    - We will have to generate an empty Block for both possible paths of the control flow

- Different architectures have different conventions for how they handle Boolean branch decisions
  - Two major approaches
    1. Branch operations allow Boolean comparison of 2 regs
       - More powerful approach
    2. Separate comparison operation, followed by a branch based on flags set by comparison
       - Used by X86
       - Other operations also set flags – have to be careful

Low-Level Code Generation

- Need to determine which branch format you will use
  - Want to keep low-level code architecture independent
- Probably the more general approach is to treat Boolean decision as part of the branch operation
  - Can easily convert to other model in subsequent pass
  - May not be as straightforward to implement
- Consider the generic control flow statement
  - It will typically generate a branch operation
    - Closes out the current basic block
  - Control can pass to one of two paths
    1. The next sequential, or “fall through” path
    2. The branch target path
  - This requires creation of 2 new blocks (in some cases 3)
    - The fall-through can be hooked up, but the other is hanging

Low-Level Code Generation

- Let's consider the if_stmt first
  - Start with one without an else block
  - We can fall through into the then block
  - We branch to the statement following the if (op2)
  - Our then block should fall through to op2 also
- Once genCode for IfStmt has generated code for the branch (and created target block for branch), it closes this Block, and creates a new currentBlock
- It then calls genCode on the ThenStmt
- When genCode for the ThenStmt returns, genCode for IfStmt closes currentBlock (note: the ThenStmt could have created multiple blocks) and links target block in

```
op1;
if (a < b) c=0;
op2;
```

```
bb0:
  op1;
  r5 = lt (r1, r2)
  branch (r5, 0), bb1
bb2:
  mov r3, 0
bb1:
  op2;
```

Low-Level Code Generation

- Well, we skipped one ugliness
  - How do we generate the branch itself, particularly the decision expression (which can be arbitrarily complex)?
    - Must handle short-circuit evaluation properly
- Key to handling Boolean expressions is that they should produce a Boolean result, i.e., a register holding either a 1 or 0
  - Our final branch (part of the if_stmt) can branch based on comparing this value to 0

- Consider expression a < b - it needs a result reg (r1)
  - Generates one of the following forms (depending on arch):

```
r1 = lt r5, r6
```

```
    mov r1, 0
    br_gte (r5, r6) bb2
    mov r1, 1
bb2:
```

Low-Level Code Generation

- Entire if_stmt would then be

  if (a < b) c=0;

  - Not real efficient code
  - Later optimization will fix

```
bb0: op1;
     r1 = lt r5, r6
     br_eq (r1, 0) bb2
bb1: mov r3, 0
bb2: op2
```

- So, overall pattern for the

  if_stmt wants to call genCode on its Expr, look at the register that its child expression used, then generate the branch based on it

- Let's think about short-circuit
  - Occurs with && and ||
  - If doing genCode on a BinaryExpression, and the BinopType is one of these, need to do some extra work

Low-Level Code Generation

- Short-circuit (cont)
  - Consider (r1 < r2) && (r3 < r4)
  - We are in genCode for BinaryExpression of type &&
  - First, call genCode on lhs child
  - The && expression must generate the following pattern

    if (lhs == 0) result = 0

    else eval_rhs; if (rhs == 1) result = 1 else result =0

```
      mov r11, 0   // result =0
      r10 = lt r1, r2     // eval lhs
      br_eq (r10, 0), bb3
bb1:  r12 = lt r3, r4   // eval rhs   // or r11  =?
      br_eq (r12, 0), bb3
bb2:  mov r11, 1
bb3:
```

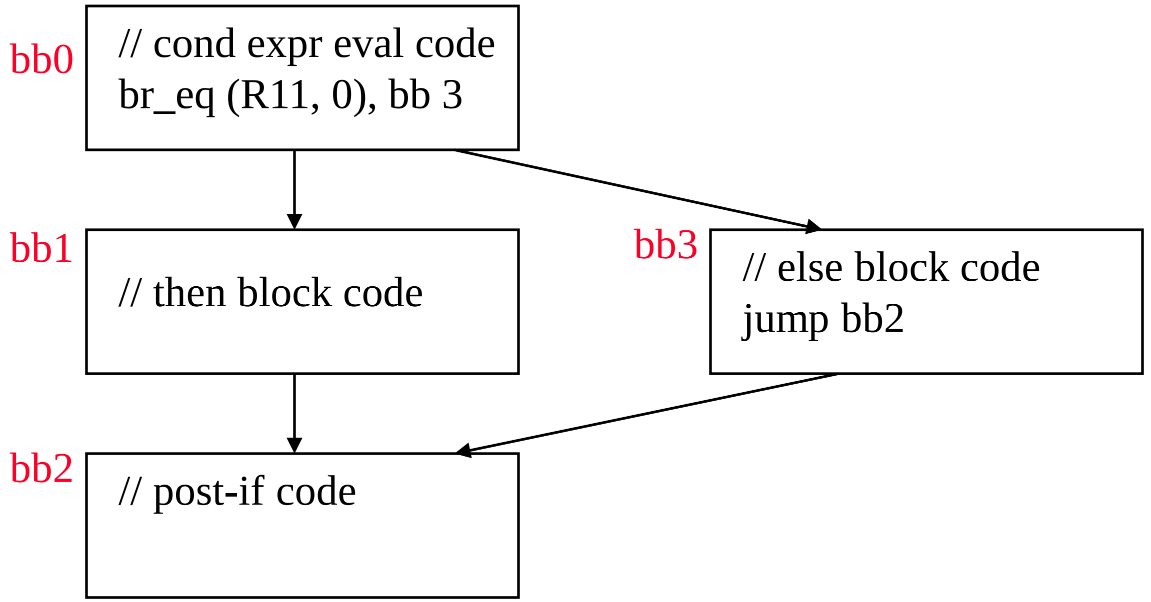
Low-Level Code Generation

- If we are generating code for if ((a<b) && (c<d) && (e<f)) the resultant code would get pretty ugly
  - But it would be the result of each sub-expression generating its own piece
  - Need to create each genCode module such that it does its thing, regardless of what nodes may be above or below it

- OK, we looked at simple if_stmt
  - Let's look at what happens if we have an if-then-else
  - The branch operation can't just jump to the statement following the if_stmt
  - How would we handled this?

Low-Level Code Generation

- If-then-else statements
  - Branch to the else block
    - Jump from there to the post-if block
  - Could reverse polarity, and jump to then, fall through to else



Low-Level Code Generation

- If-then-else statements (cont)
  - Some minor challenges
    - genCode for an IfStmt needs to create 3 Blocks
      - Then block
      - Else block
      - Post-if block
    - These blocks need to get passed to appropriate genCode routines
    - Note that the Then block and Post-if blocks can be added sequentially to the current list of blocks
      - However, the Else block sort of hangs out in space
      - Need to create list of hanging blocks like this, which will eventually be appended to the main list of blocks
  - *Hey, I never said Project #3 was trivial !!!*

Low-Level Code Generation

- If you've figured out if statements, while loops and for loops should be pretty easy
- Consider the while loop
  - Very similar to the if without an else
  - Branch at beginning jumps to post-while block
  - Two options
    - Add an unconditional jump at end of while body, back to beginning of code used to test entry into the loop
      - Simplest and least code
    - Repeat the test code again at the end of the loop body, but reverse polarity of branch back to start of body
      - Most efficient (don't pay for unconditional jump)

Low-Level Code Generation

- What about for loops?
  - A for loop is identical to a while loop from the compiler's perspective
  - Simply generate extra code involved in for loop, and do everything else same as while loop
    - Put i++ just before jump back to test
    - Maybe use common subroutine to generate code

```
for (i = 0; i < n; i++) {
    body
}
```

=

```
i = 0;
while (i < n) {
    body
    i++
}
```

Low-Level Code Generation

- We return now to the subject of generating code for data references
  - First key concept we've already referred to is the idea of promoting variables to register
    - Assume that you have an infinite number of virtual registers (both integer and floating point)
    - Simply assume that all local variables will live and die in register, and that no memory location needs to be allocated
      - Register allocator will create memory space if it can't fit all virtual registers into physical registers
    - Global variables live between functions, and must have a home memory location allocated
      - Can load into register, use it from there, and then store back to memory at end of function
      - Typically just used from memory, and then optimizer makes better use of registers

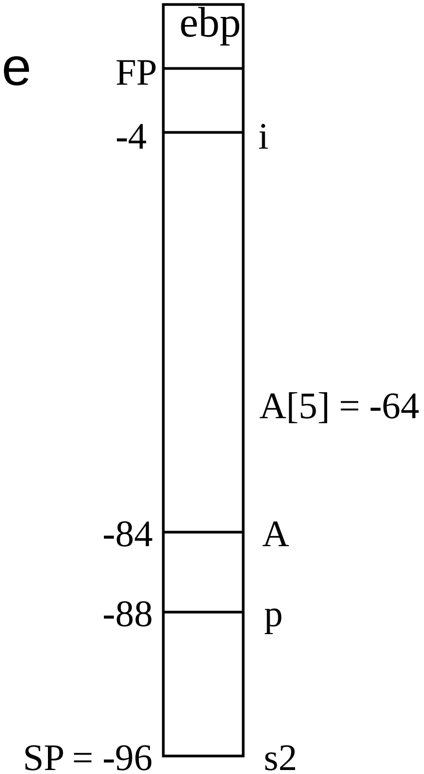
Low-Level Code Generation

- So, all local scalar values which can fit in memory are allocated to register
  - Exception: if &x ever done or declared static
- Objects, structs, pointers, and arrays are a bit more problematic
  - Pointers
    - Store pointer in register, then do load (R1) to dereference
  - Arrays
    - Compute allocation size, and declare in global area or make room on stack frame
    - A[i]   = load (&A + type_size * i)
    - Address of A[6] can be computed at compile time, either as offset from global label (beginning of array) or from FP/SP

Low-Level Code Generation

- More complex data references (cont)
  - structs
    - At compile time, can compute total size required
      - Take into account data alignment
    - At compile time, can compute offset into the struct for any particular field
    - A.r = load (&A + offset of field r)
    - Address will be an offset from either a global location or the FP/SP

- Example code containing pointers, arrays, structs



Low-Level Code Generation

- As you can see, the compiler just decides where data goes and creates the necessary operations to access it
- Let's look briefly at objects
  - Dynamically allocated in java
  - C++ can allocate to DS or stack frame unless explicitly made dynamic
  - Similar to structs, fields of object are simply offsets from object pointer
    - If we have p as a struct pointer, find p.a by adding (&p + offset a) and then doing load
    - Same thing if p is an object, and we reference a field of that object
- Object example

Instruction Selection

- The Code Generator frequently has options of how it will generate code
  - e.g., using r1 = r1 + 1   or incr r1
  - RISC vs CISC instructions
- In general, the Code Generator won’t make decisions on whether to use increment operations
  - A later, architecture-specific phase will decide that
- However, the Code Generator needs to decide whether to generate RISC or CISC instructions
  - Most architectures designed as load-store architectures
    - Designed not to use complicated instructions
  - For old X86 architectures, it didn’t matter which version you chose – both take same time – CISC might take less space
  - Newer superscalar X86 architectures will perform much better it the load-store subset of the instruction set is used
- Bottomline: Code Generator should choose RISC subset