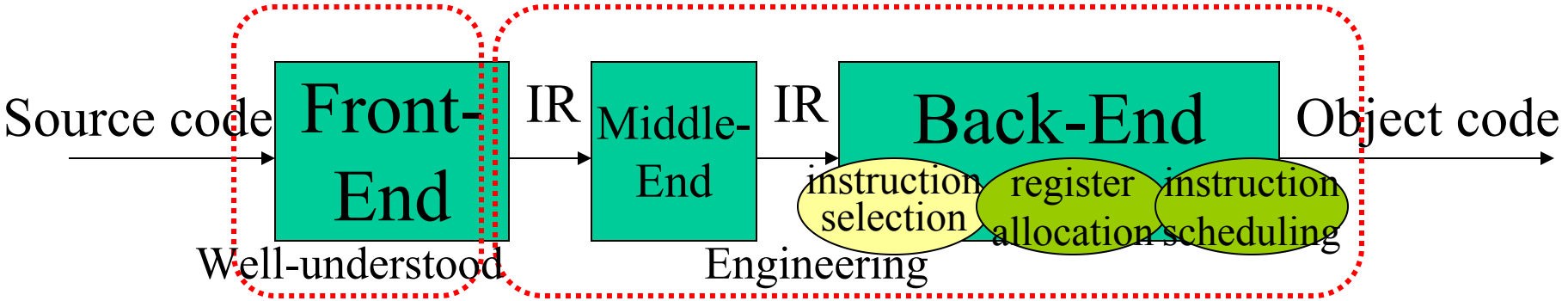
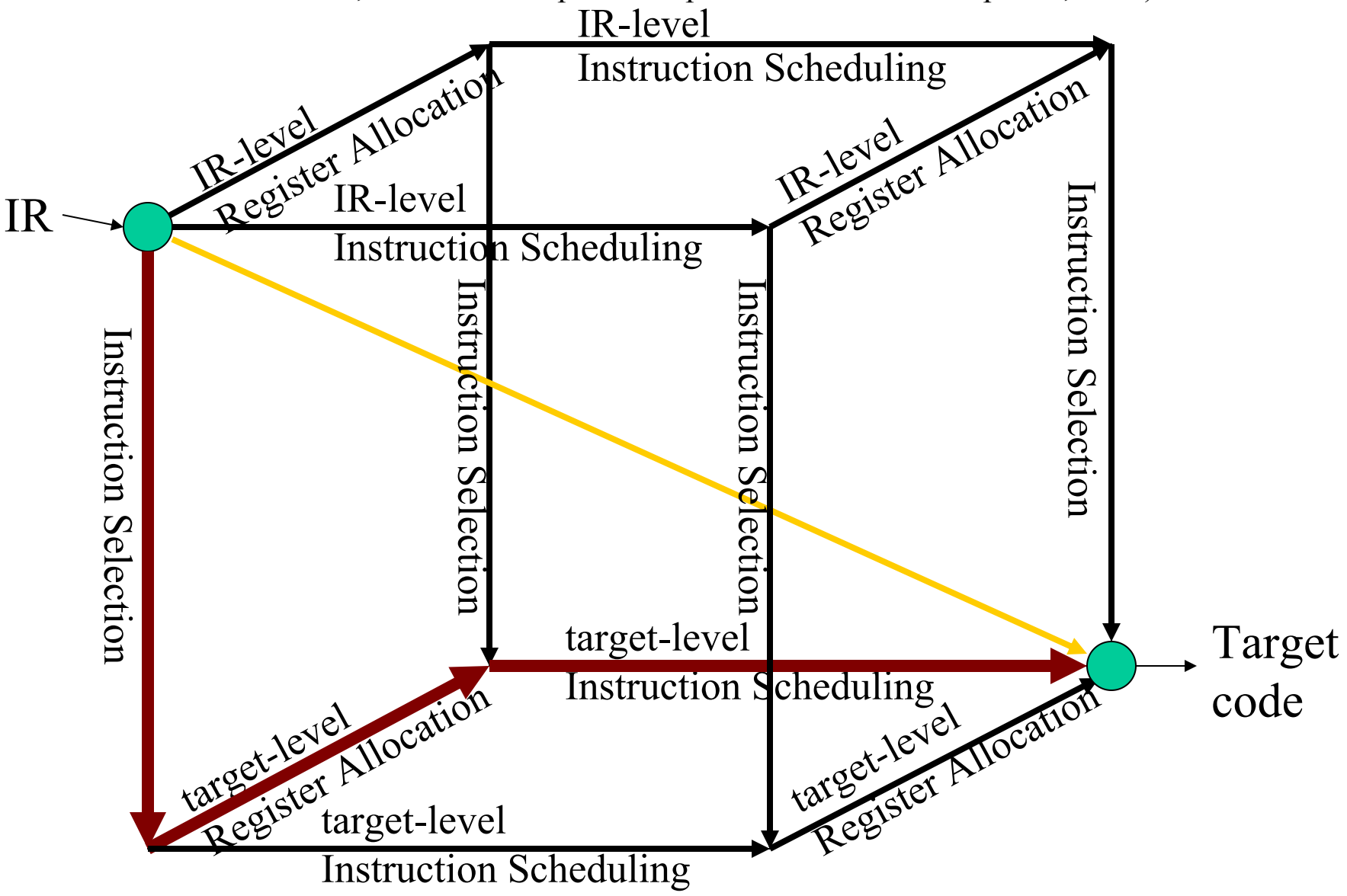
Lecture 15: Code Generation - Instruction Selection



- Last Lecture:
  - The Procedure Abstraction
  - Run-Time Organisation: Activation records for procedures.
- Where are we?
  - Code generation: we assume the following model:
    - instruction selection: mapping IR into assembly code
    - register allocation: decide which values will reside in registers.
    - instruction scheduling: reorder operations to hide latencies…
  - Conventional wisdom says that we lose little by solving these (NP-complete) problems independently.

Code Generation as a phase-decoupled problem

(thanks for inspiration to: C.Kessler, A.Bednarski; Optimal Integrated Code Generation for VLIW architectures; 10th Workshop on Compilers for Parallel Computers, 2003)



Instruction Selection

- Characteristics:
  - use some form of pattern matching
  - can make locally optimal choices, but global optimality is NP-complete
  - assume enough registers.
- Assume a RISC-like target language
- Recall:
  - modern processors can issue multiple instructions at the same time.
  - instructions may have different **latencies**: e.g., add: 1 cycle; load 1-3 cycles; imult: 3-16 cycles, etc
- Instruction Latency: length time elapsed from the time the instruction was issued until the time the results can be used.

Our (very basic) instruction set

- load r1, @a ; load register r1 with the memory value for a
- load r1, [r2] ; load register r1 with the memory value pointed by r2
- mov r1, 3 ; r1=3
- add r1, r2, r3 ; r1=r2+r3

  (same for mult, sub, div)
- shr r1, r1, 1 ; r1=r1>>1 (shift right; r1/2)
- store r1 ; store r1 in memory
- nop ; no operation

Code generation for arithmetic expressions

- Adopt a simple treewalk scheme; emit code in postorder:

```
expr(node)
{ int result, t1, t2;
  switch(type(node))
  { case *,/,+,-:
      t1=expr(left child(node));
      t2=expr(right child(node));
      result=NextRegister();
      emit(op(node),result,t1,t2);
      break;
    case IDENTIFIER:
      t1=base(node); t2=offset(node);
      result=NextRegister();
      emit(…) /* load IDENTIFIER */
      break;
    case NUM:
      result=NextRegister();
      emit(load result, val(node));
      break;
  }
  return result;
}
```

Example: x+y:

```
load r1, @x
load r2, @y
add r3,r1,r2
```

(`load r1,@x` would involve a load from address base+offset)

Issues with arithmetic expressions

- What about values already in registers?
  - Modify the IDENTIFIER case.
- Why the left subtree first and not the right?
  - (cf. 2*y+x; x-2*y; x+(5+y)*7): the most demanding (in registers) subtree should be evaluated first (this leads to the Sethi-Ullman labelling scheme – first proposed by Ershov – see Aho2 §8.10 or Aho1 §9.10).
- 2nd pass to minimise register usage/improve performance.
- The compiler can take advantage of commutativity and associativity to improve code (but not for floating-point operations)

Issues with arithmetic expressions - 2

- Observation: on most processors, the cost of a mult instruction might be several cycles; the cost of shift and add instructions is, typically, 1 cycle.
- Problem: generate code that multiplies an integer with an unknown using only shifts and adds!
- E.g.:

  325*x = 256*x+64*x+4*x+x or (4*x+x)*(64+1) (Sparc)
- For division, say x/3, we could compute 1/3 and perform a multiplication (using shifts and adds)… but this is getting complex!

  [For the curious only: see the "Hacker's delight" and http://www.hackersdelight.org/divcMore.pdf]

Trading register usage with performance

- Example: w=w*2*x*y*z

```
 1. load r1, @w
 2. mov  r2, 2
 6. mult r1,r1,r2
 7. load r2, @x
12. mult r1,r1,r2
13. load r2, @y
18. mult r1,r1,r2
19. load r2, @z
24. mult r1,r1,r2
26. store r1
```

Hardware will stall until result is available!

```
 1. load r1, @w   ; load: 5 cycles
 2. load r2, @x
 3. load r3, @y
 4. load r4, @z
 5. mov  r5, 2    ; mov: 1 cycle
 6. mult r1,r1,r5 ; mult: 2 cycles
 8. mult r1,r1,r2
10. mult r1,r1,r3
12. mult r1,r1,r4
14. store r1      ; store: 5 cycles
```

2 registers, 31 cycles

5 registers, 19 cycles

- Instruction scheduling (the problem): given a code fragment and the latencies for each operation, reorder the operations to minimise execution time (produce correct code; avoid spilling registers)

Array references

- Agree to a storage scheme:
  - Row-major order: layout as a sequence of consecutive rows: A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]
  - Column-major order: layout as a sequence of consecutive columns: A[1,1], A[2,1], A[1,2], A[2,2], …
  - Indirection vectors: vector of pointers to pointers to … values; best storage is application dependent; not amenable to analysis.
- Referencing an array element, where w=sizeof(element):
  - row-major, 2d: base+((i$_1$–low$_1$)*(high$_2$–low$_2$+1)+i$_2$–low$_2$)*w
  - column-major, 2d: base+((i$_2$–low$_2$)*(high$_1$–low$_1$+1)+i$_1$–low$_1$)*w
  - general case for row-major order:
    - ((…(i$_1$n$_2$+i$_2$)n$_3$+i$_3$)…)n$_k$+i$_k$)*w+base - w*((…((low$_1$*n$_2$)+low$_2$)n$_3$+low$_3$)…)n$_k$+low$_k$), where n$_i$=high$_i$–low$_i$+1

Boolean and relational values

expr→ not or-term | or-term
or-term→or-term or and-term | and-term
and-term→and-term and boolean | boolean
boolean→true | false | rel-term
rel-term→rel-term rel-op expr | expr
rel-op→ < | > | == | != | >= | <=

- Evaluate using treewalk-style generation. Two approaches for translation: numerical representation (0/1) or positional encoding.
- B or C and not D: r1=not D; r2=r1 and C; r3=r2 or B.
- if (a<b) then … else… : comp rx,ra,rb; br rx L1, L2; L1: …code for then...; br L3; L2: …code for else...; L3: ...
- Short-circuit evaluation: the C expression (x!=0 && y/x>1) relies on short-circuit evaluation for safety.

Control-Flow statements

- If expr then stmt1 else stmt2
  - 1. Evaluate the expr to true or false
  - 2. If true, fall through to then; branch around else
  - 3. If false, branch to else; fall through to next statement.

    (if not(expr) br L1; stmt1; br L2; L1: stmt2; L2: …)
- While loop; for loop; or do loop:
  - 1. Evaluate expr
  - 2. If false, branch beyond end of loop; if true, fall through
  - 3. At end, re-evaluate expr
  - 4. If true, branch to top of loop body; if false, fall through

    (if not(expr) br L2; L1: loop-body; if (expr) br L1; L2: …)
- Case statement: evaluate; branch to case; execute; branch
- Procedure calls: recall last lecture...

Conclusion

- (Initial) code generation is a pattern matching problem.
- Instruction selection (and register allocation) was the problem of the 70s. With the advent of RISC architectures (followed by superscalar and superpipelined architectures and with the ratio memory access vs CPU speed starting to rise) the problems shifted to register allocation and instruction scheduling.

- Reading: Aho2, Chapter 8 (more information than covered in this lecture but most of it useful for later parts of the module); Aho1 pp.478-500; Hunter, pp.186-198; Cooper, Chapter 11 (NB: Aho treats intermediate and machine code generation as two separate problems but follows a low-level IR; things may vary in reality).

- Next time: Register allocation