18-747 Lecture 16: Scheduling and Register Allocation

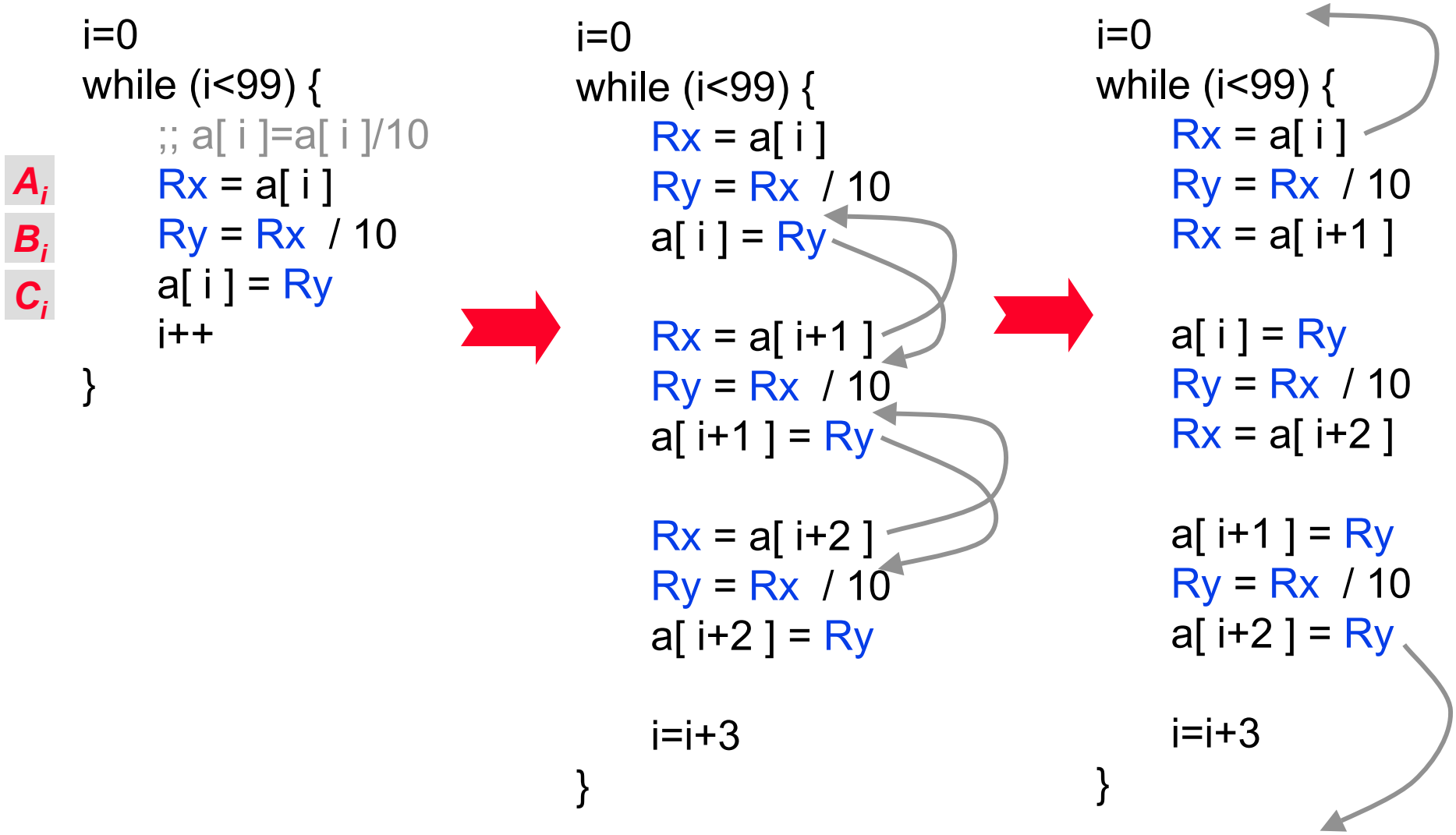
James C. Hoe
Dept of ECE, CMU
October 29, 2001

Reading Assignments: MJ Ch10 & Ch11

Announcements: Quiz 1 re-grade requests due Wednesday 10/31

Handouts:

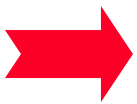
Software Pipelining



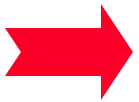
Software Pipelining *(continued)*

A_i
B_i
C_i

```
i=0
while (i<99) {
    ;; a[ i ]=a[ i ]/10
    Rx = a[ i ]
    Ry = Rx  / 10
    a[ i ] = Ry
    i++
}
```

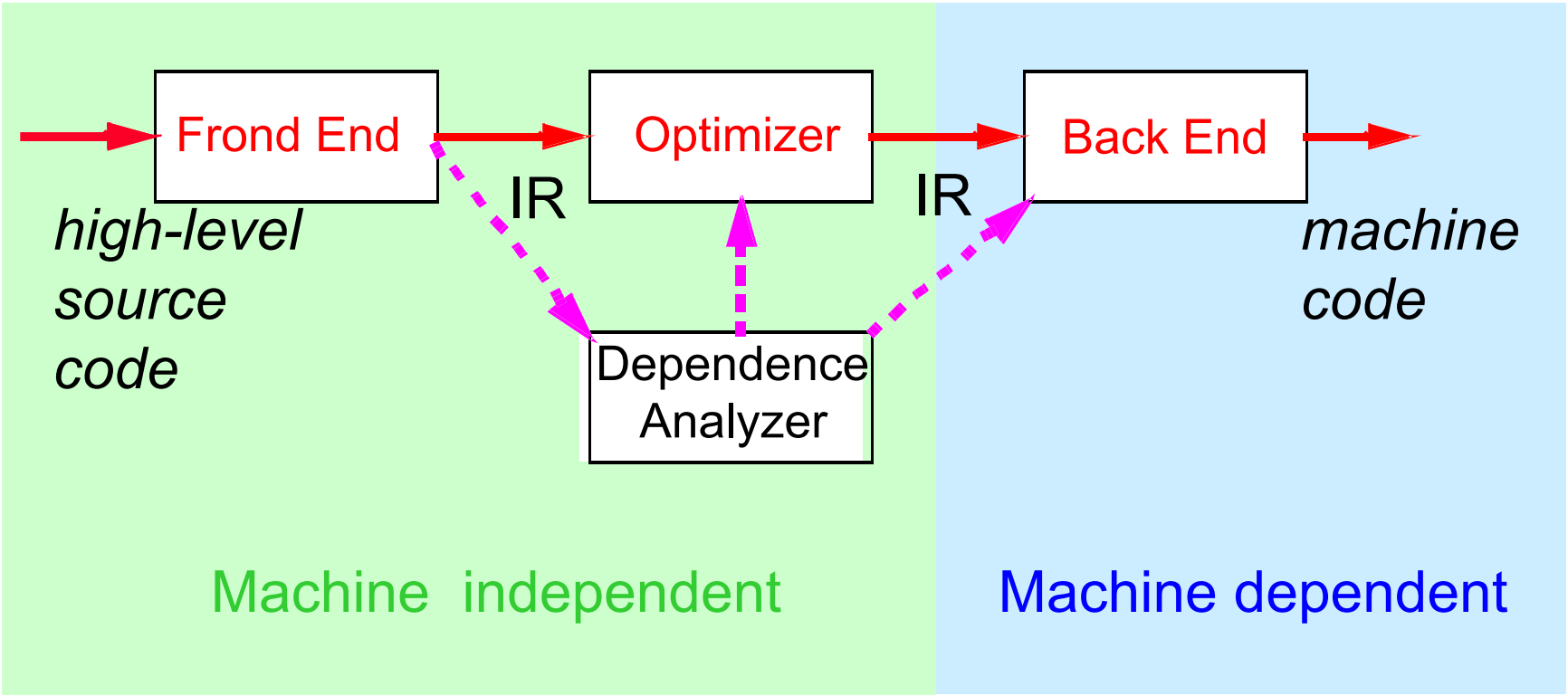


```
i=0
while (i<99) {
    Rx = a[ i ]
    Ry = Rx  / 10
    a[ i ] = Ry

    Rx = a[ i+1 ]
    Ry = Rx  / 10
    a[ i+1 ] = Ry

    Rx = a[ i+2 ]
    Ry = Rx  / 10
    a[ i+2 ] = Ry

    i=i+3
}
```



```
i=0
Ry=a[ 0 ] / 10
Rx=a[ 1 ]

while (i<97) {
    a[i]=Ry          C_i

    Ry=Rx / 10  B_i+1

    Rx=a[i+2]    A_i+2

    i++
}

a[97]=Ry
a[98]=Rx / 10
```

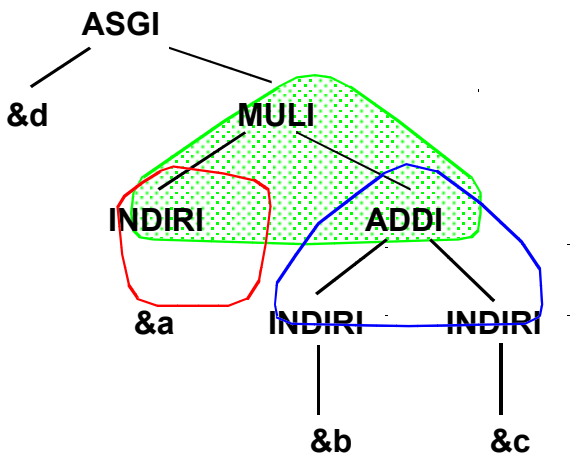
Compiler Structure



(IR= intermediate representation)

Code Selection

- Map IR to machine instructions (e.g. pattern matching)

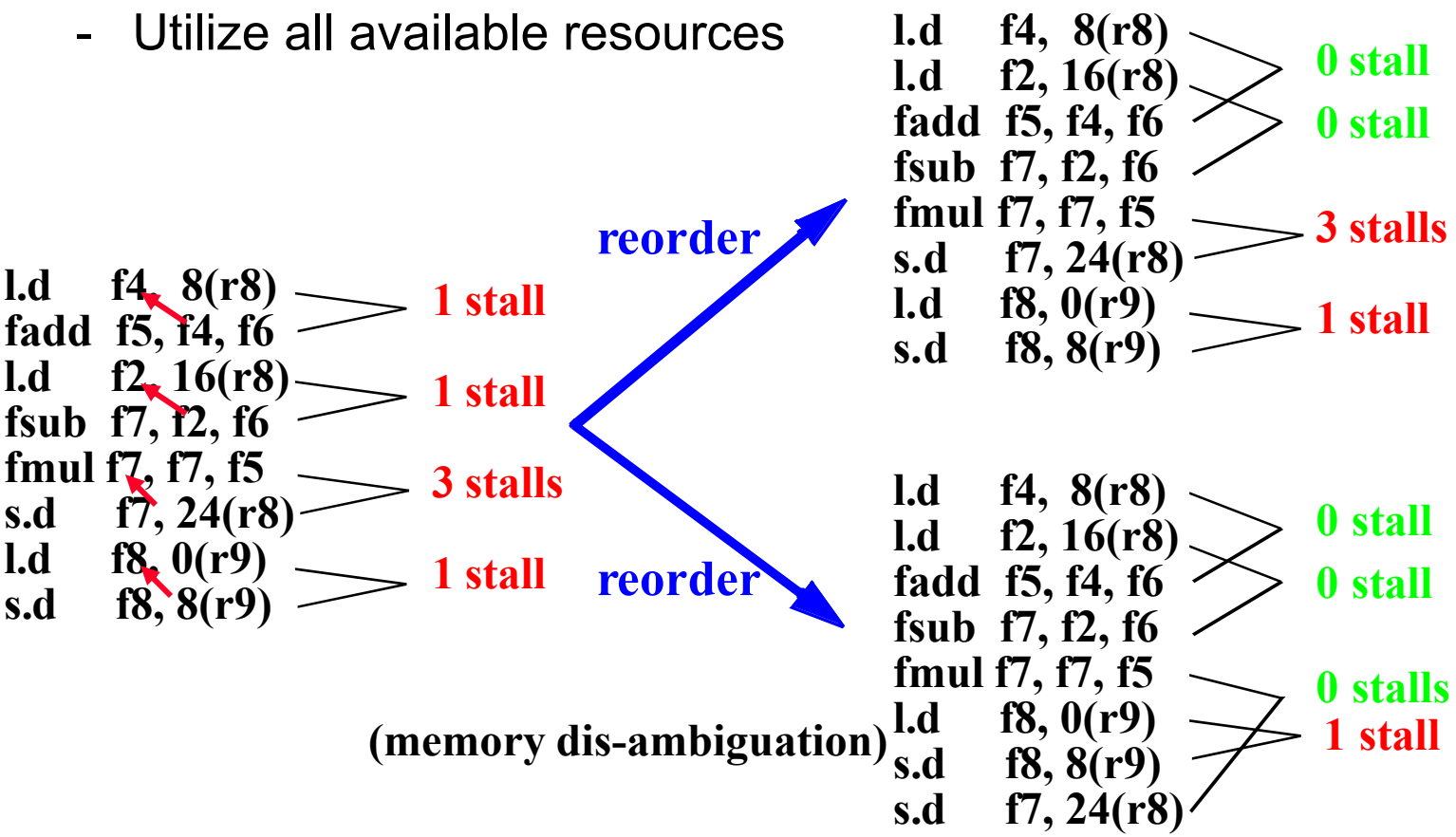


addi Rt1, Rb, Rc
muli Rt2, Ra, Rt1

```
Inst *match (IR *n) {
    switch (n->opcode) {
        case ……..:
        case MUL :
            l = match (n->left());
            r = match (n->right());
             if (n->type == D || n->type == F )
                 inst = mult_fp( (n->type == D), l, r );
            else
                 inst = mult_int ( (n->type == I), l, r);
            break;
        case ADD :
            l = match (n->left());
            r = match (n->right());
             if (n->type == D || n->type == F)
                 inst = add_fp( (n->type == D), l, r);
            else
                 inst = add_int ((n->type == I), l, r);
            break;
          case ……..:
    }
    return inst;
}
```

Code Scheduling

- Rearrange code sequence to minimize execution time
  - Hide instruction latency
  - Utilize all available resources

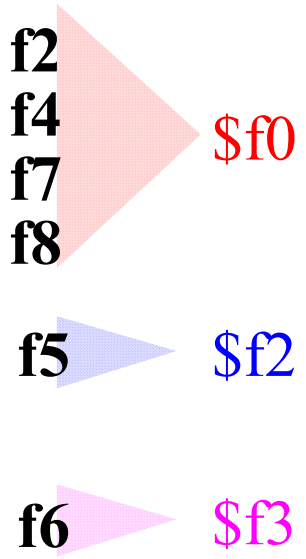


Register Allocation

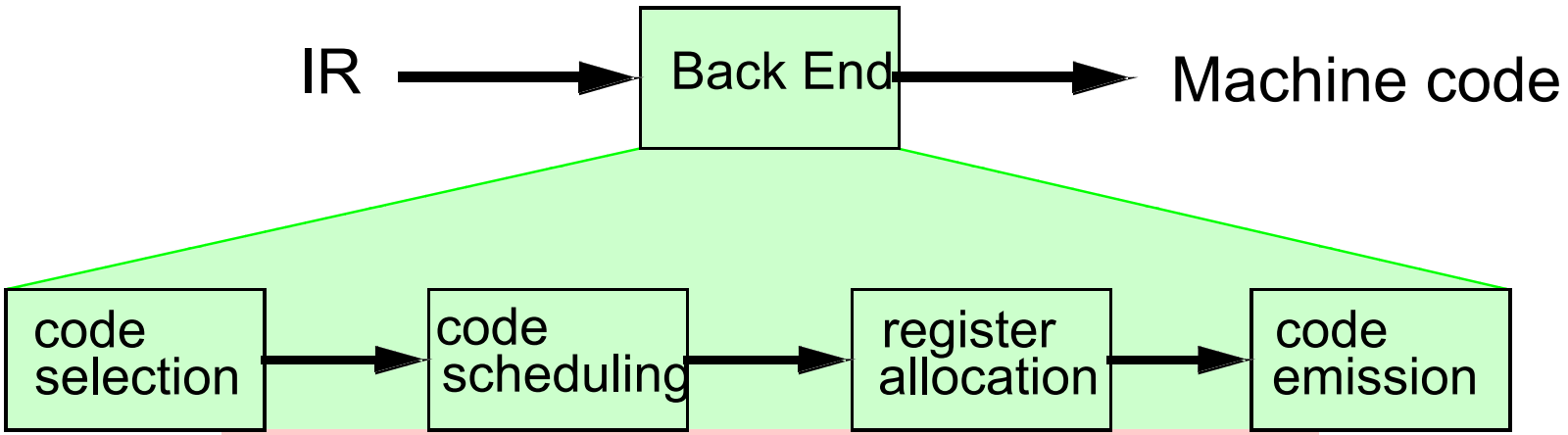
- Map virtual registers into physical registers
  - minimize register usage to reduce memory accesses
  - but introduces false dependencies . . . . .

```
l.d    f4,  8(r8)
fadd   f5, f4, f6
l.d    f2, 16(r8)
fsub   f7, f2, f6
fmul   f7, f7, f5
s.d    f7, 24(r8)
l.d    f8, 0(r9)
s.d    f8, 8(r9)
```

```
l.d    $f0,  8(r8)
fadd   $f2, $f0, $f3
l.d    $f0, 16(r8)
fsub   $f0, $f0, $f3
fmul   $f0, $f0, $f2
s.d    $f0, 24(r8)
l.d    $f0, 0(r9)
s.d    $f0, 8(r9)
```



Back End



- map virtual registers into architect registers
- rearrange code
- target machine specific optimizations
  - delayed branch
  - conditional move
  - instruction combining
    - auto increment addressing mode
    - add carrying (PowerPC)
    - hardware branch (PowerPC)

Instruction-level IR

Code Scheduling

- Objectives: minimize execution latency of the program
  - Start as early as possible instructions on the critical path
  - Help expose more instruction-level parallelism to the hardware
  - Help avoid resource conflicts that increase execution time
- Constraints
  - Program Precedences
  - Machine Resources
- Motivations
  - Dynamic/Static Interface (DSI): By employing more software (static) optimization techniques at compile time, hardware complexity can be significantly reduced
  - Performance Boost: Even with the same complex hardware, software scheduling can provide additional performance enhancement over that of unscheduled code

Precedence Constraints

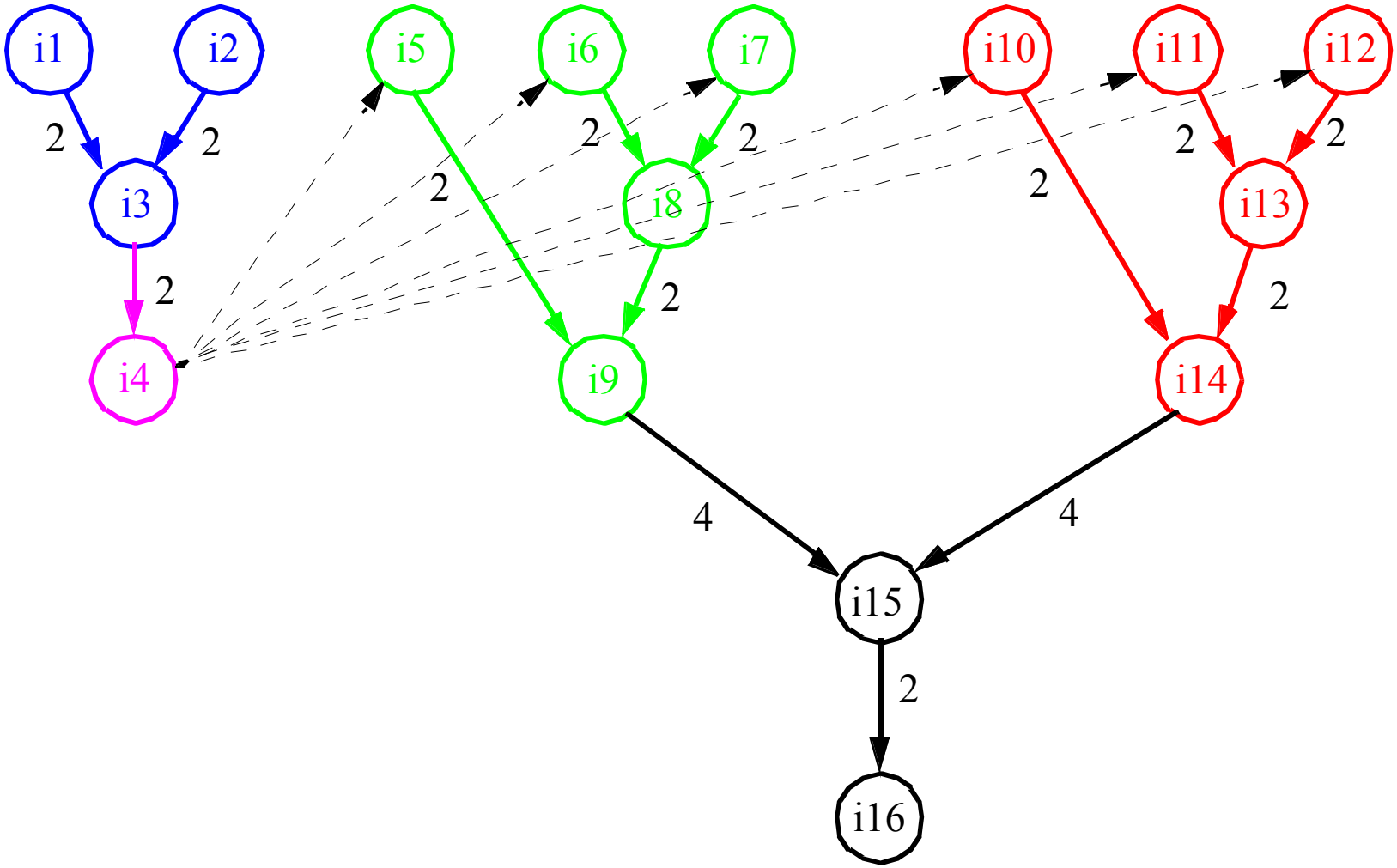
- Minimum required ordering and latency between definition and use
- Precedence graph
  - Nodes: instructions
  - Edges (a→b): a precedes b
  - Edges are annotated with minimum latency

```
w[i+k].ip = z[i].rp + z[m+i].rp;
w[i+j].rp = e[k+1].rp*
            (z[i].rp -z[m+i].rp) -
            e[k+1].ip *
            (z[i].ip - z[m+i].ip);
```

FFT code fragment

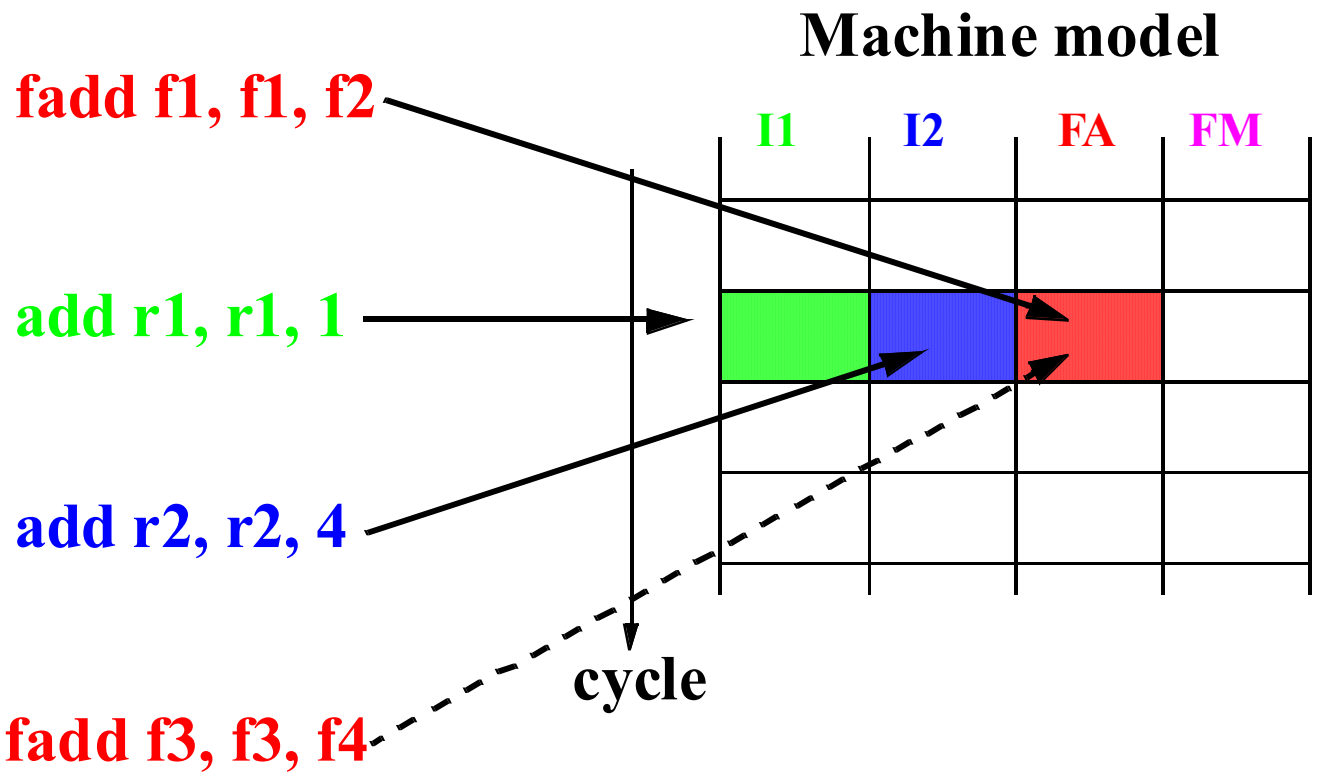
```
i1: l.s f2, 4(r2)
i2: l.s f0, 4(r5)
i3: fadd.s f0, f2, f0
i4: s.s f0, 4(r6)
i5: l.s f14, 8(r7)
i6: l.s f6, 0(r2)
i7: l.s f5, 0(r3)
i8: fsub.s f5, f6, f5
i9: fmul.s f4, f14, f5
i10: l.s f15, 12(r7)
i11: l.s f7, 4(r2)
i12: l.s f8, 4(r3)
i13: fsub.s f8, f7, f8
i14: fmul.s f8, f15, f8
i15: fsub.s f8, f4, f8
i16: s.s f8, 0(r8)
```

Precedence Graph



Resource Constraints

- Bookkeeping
  - Prevent resources from being oversubscribed



List Scheduling for Basic Blocks

1. Assign priority to each instruction
2. Initialize ready list that holds all ready instructions

   Ready = data ready and can be scheduled
3. Choose one ready instruction *I* from ready list with the highest priority

   Possibly using tie-breaking heuristics
4. Insert *I* into schedule

   Making sure resource constraints are satisfied
5. Add those instructions whose precedence constraints are now satisfied into the ready list

Priority Functions/Heuristics

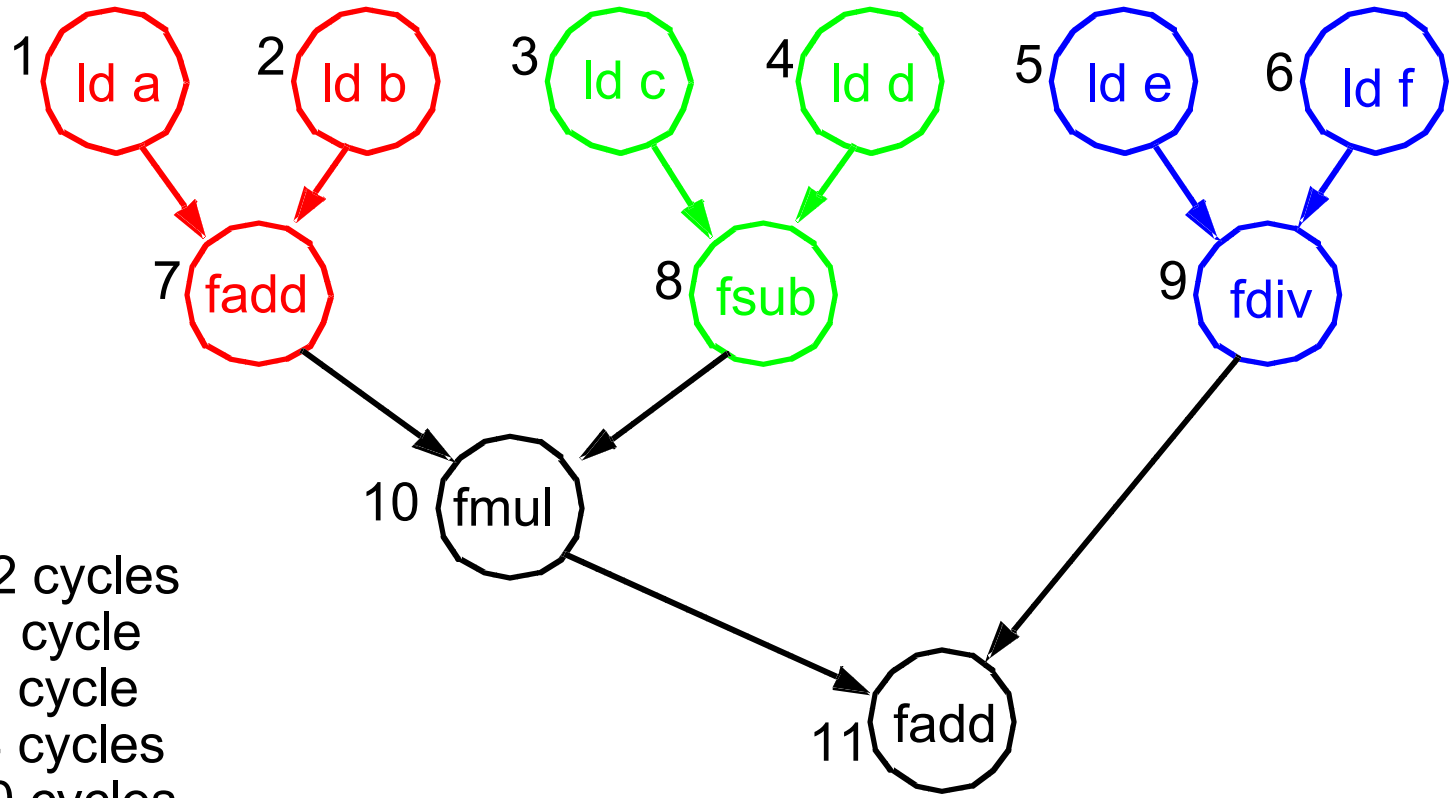
- Number of descendants in precedence graph
- Maximum latency from root node of precedence graph
- Length of operation latency
- Ranking of paths based on importance
- Combination of above

Orientation of Scheduling

- Instruction Oriented
  - Initialization (priority and ready list)
  - Choose one ready instruction *I* and find a slot in schedule
    make sure resource constraint is satisfied
  - Insert *I* into schedule
  - Update ready list
- Cycle Oriented
  - Initialization (priority and ready list)
  - Step through schedule cycle by cycle
  - For the current cycle *C*, choose one ready instruction *I*
    be sure latency and resource constraints are satisfied
  - Insert *I* into schedule (cycle *C*)
  - Update ready list

List Scheduling Example

(a + b) * (c - d) + e/f



load: 2 cycles
add: 1 cycle
sub: 1 cycle
mul: 4 cycles
div: 10 cycles

orientation: cycle
direction: backward
heuristic: maximum latency to root

Scalar Scheduling Example

Cycle	Ready list	Schedule	Code
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

Scalar Scheduling Example

Cycle	Ready list	Schedule	Code
1	6	6	ld f
2	5 6	5	ld e
3	4 5 6	4	ld d
4	4 9	9	fdiv (e/f)
5	3 4 9	3	ld c
6	2 3 4 9	2	ld b
7	1 2 3 4 9	1	ld a
8	1 2 8 9	8	fsub (c – d)
9	7 8 9	7	fadd (a + b)
10	9 10	10	fmul
11	9 10		nop
12	9 10		nop
13	9 10		nop
14	11	11	fadd

green *means candidate and ready*
red means candidate but not yet ready

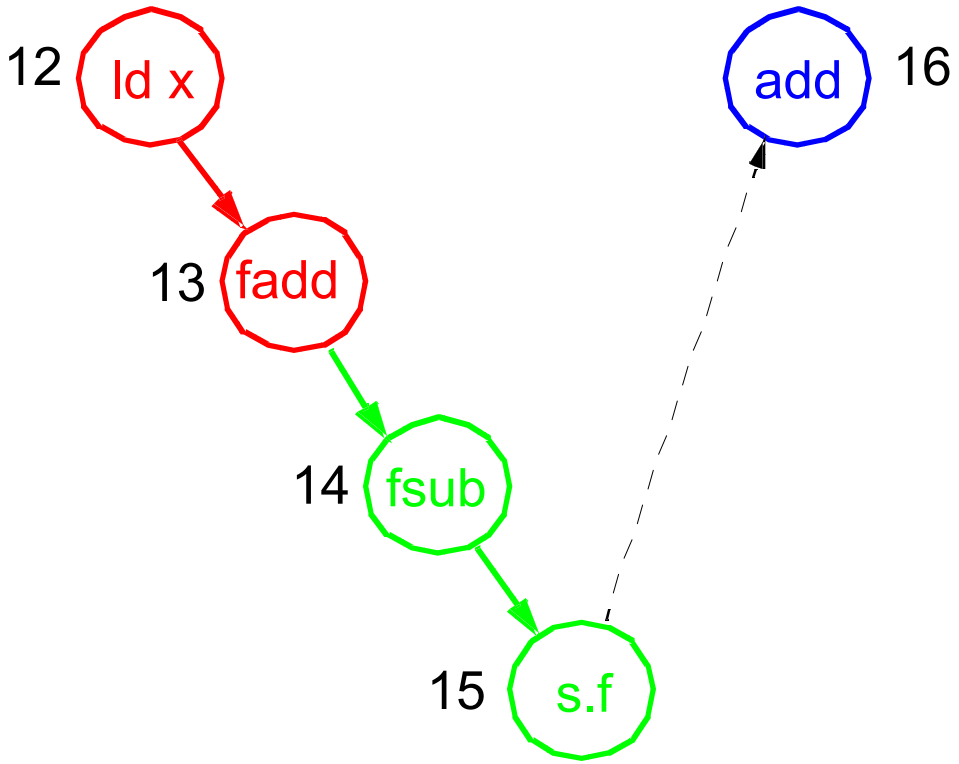
Superscalar Scheduling Example

Cycle	Ready list	Schedule	Resources			Code	
			I	F	FD		
1	6	6	X			ld f	
2	5 6	5	X			ld e	
3	5 6						
4	4 9	4 9	X		X	fdiv (e/f)	ld d
5	3 4 9	3	X			ld c	
6	2 3 4 9	2	X			ld b	
7	1 2 3 4 9	1	X			ld a	
8	1 2 8 9	8		X		fsub (c – d)	
9	7 8 9	7		X		fadd (a + b)	
10	9 10	10		X		fmul	
11	9 10					nop	
12	9 10					nop	
13	9 10					nop	
14	11	11		x		fadd	

Take Home Example

Append the following to the previous example:

```
*(p) = (x + Ry) - Rz ;
p = p + 4 ;
```



Take Home Example

Cycle	Ready list	Schedule	Resources			Code	
			I	F	FD		
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							

Take Home Example

Cycle	Ready list	Schedule	Resources			Code	
			I	F	FD		
1	6	6	X			ld f	
2	5 6	5	X			ld e	
3	5 6						
4	4 9	4 9	X		X	fdiv (e/f)	ld d
5	3 4 9	3	X			ld c	
6	2 3 4 9	2	X			ld b	
7	1 2 3 4 9	1	X			ld a	
8	1 2 8 9	8		X		fsub (c – d)	
9	7 8 9 12	7 12	X	X		fadd (a + b)	ld x
10	9 10 12	10		X		fmul	
11	9 10 13	13		X		fadd	
12	9 10 14	14		X		fsub	
13	9 10 15	15	X			s.f	
14	11 16	11 16	X	X		fadd	add

Directions of List Scheduling

- Backward Direction
  - Start with consumers of values
  - Heuristics
    - Maximum latency from root
    - Length of operation latency

    produces results just in time
- Forward Direction
  - Start with producers of values
  - Heuristics
    - Maximum latency from root
    - Number of descendants
    - Length of operation latency

    produces results as early as possible

Limitations of List Scheduling

- Cannot move instructions past conditional branch instructions in the program (scheduling limited by basic block boundaries)
- Problem: Many programs have small numbers of instructions (4-5) in each basic block. Hence, not much code motion is possible
- Solution: Allow code motion across basic block boundaries.
  - Speculative Code Motion: “jumping the gun”
    - execute instructions before we know whether or not we need to
    - utilize otherwise idle resources to perform work which we speculate will need to be done
  - Relies on program profiling to make intelligent decisions about speculation

Register Allocation

- Mapping symbolic (virtual) registers used in IR onto architected (physical) registers
- IR assumes unlimited number of symbolic registers
- If the number of "live" values is greater than the number of physical registers in a machine, then some values must be kept in memory, i.e. we must insert spill code to "spill" some variables from registers out to memory and later reload them when needed

  Imagine if you only had one register
- The objective in register allocation is to try to *maximize keeping temporaries in registers* and *minimize memory accesses (spill code)*

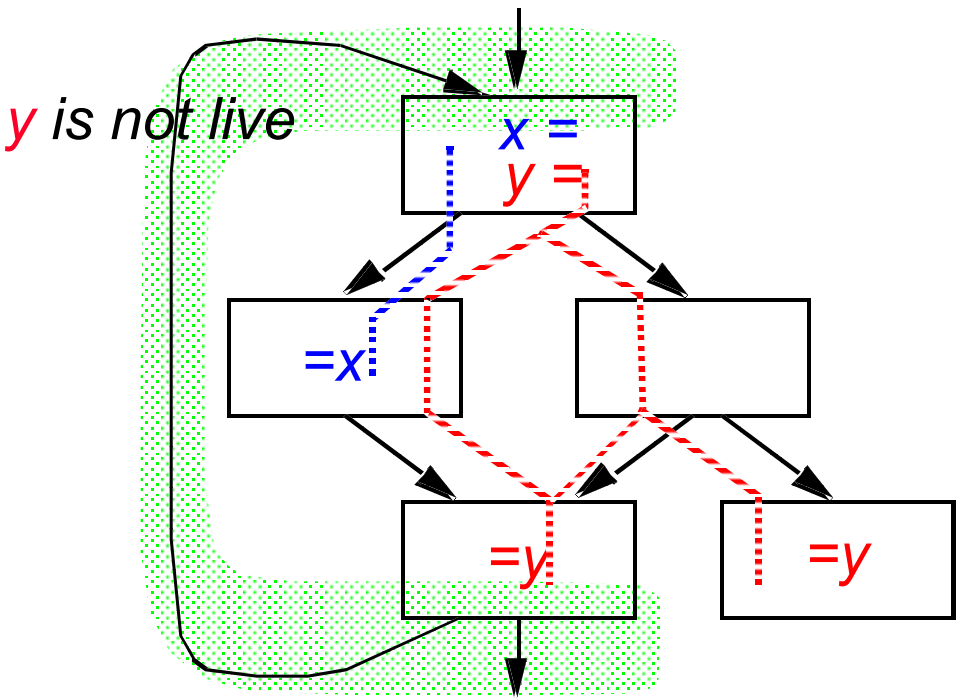
  $\Rightarrow$ *maximize register reuse*

Variable Live Range

- A variable or value is "live", along a particular control-flow path, from its definition to last use without any intervening redefinition

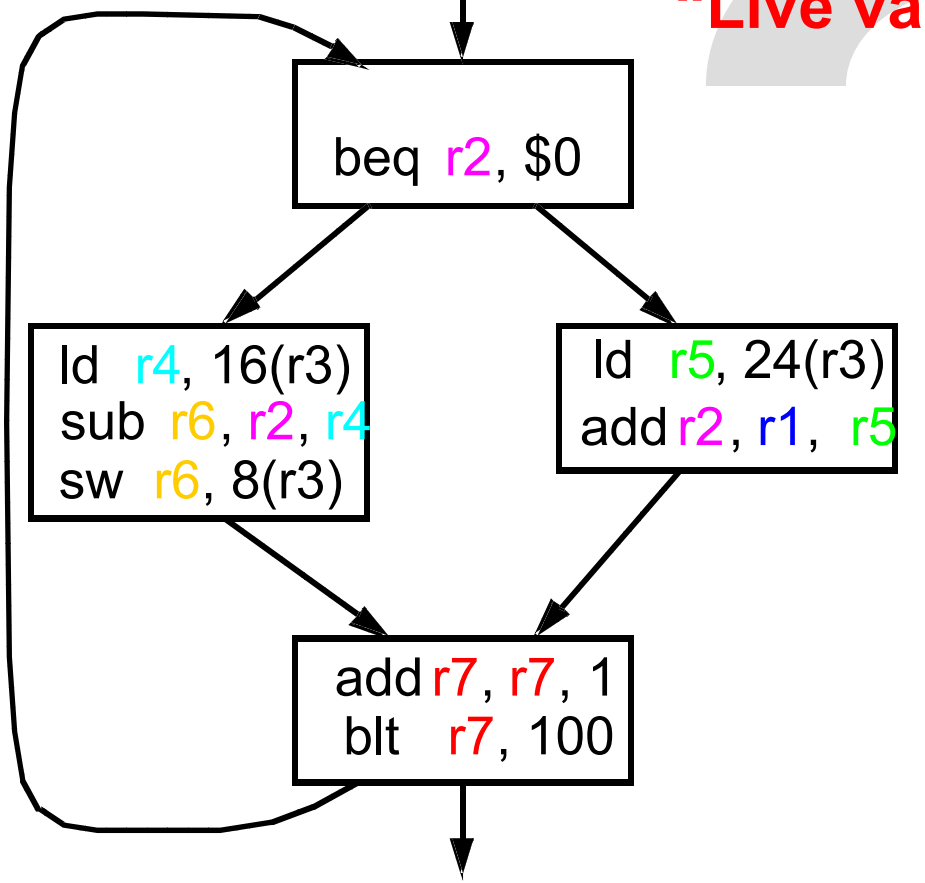
- Live range, *lr*(x)
  - Group of paths in which *x* is live
- Interference
  - *x* and *y* interfere if *x* and *y* are ever simultaneously alive along some control flow path

    i.e. $lr(x) \cap lr(y)$



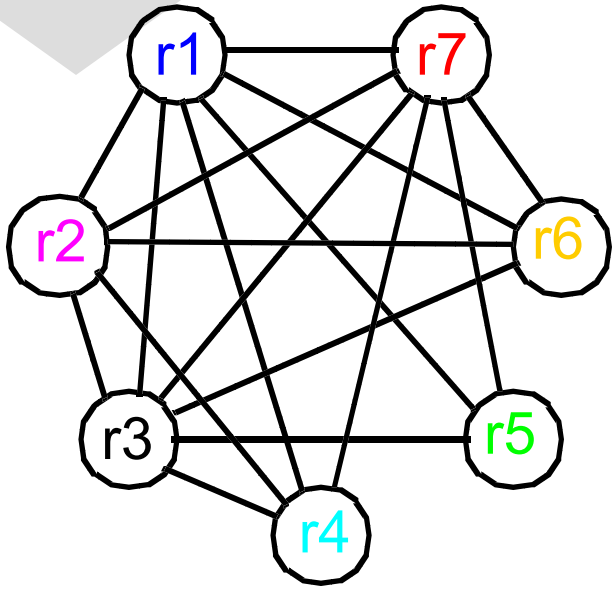
Interference Graph

r1, r2 & r3 are live-in



r1& r3 are live-out

"Live variable analysis"



Nodes: live ranges

Edges: interference

Register Interference & Allocation

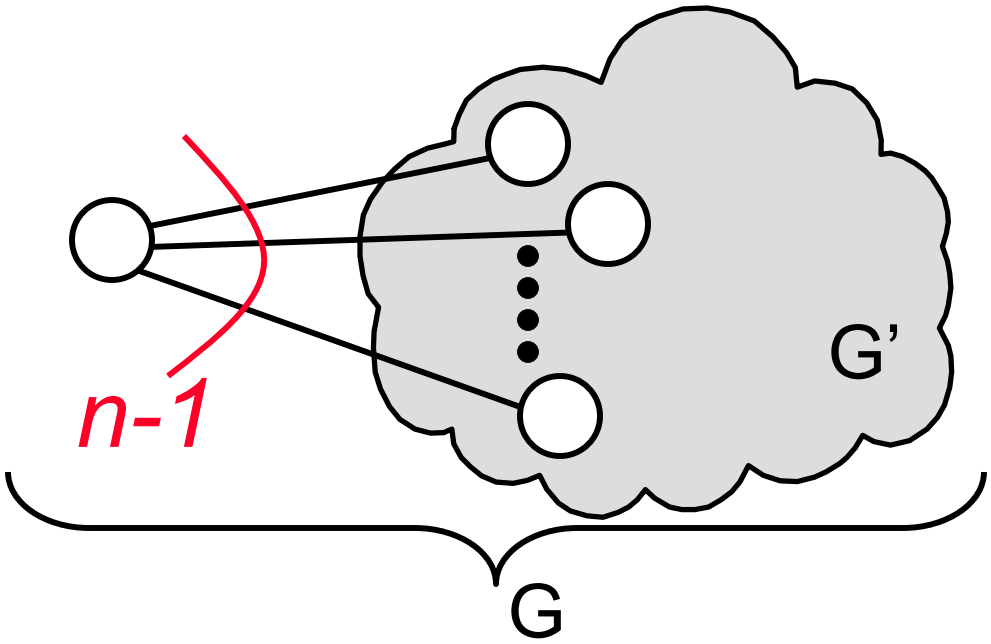
- Interference Graph: G = <E,V>
  - Nodes (V) = variables, (more specifically, their live ranges)
  - Edges (E) = interference between variable live ranges
- Graph Coloring (vertex coloring)
  - Given a graph, G=<E,V>, assign colors to nodes (V) so that **no** two adjacent (connected by an edge) nodes have the same color
  - A graph can be "*n*-colored" if no more than *n* colors are needed to color the graph.
  - The chromatic number of a graph is *min{n}* such that it can be *n*-colored
  - *n*-coloring is an NP-complete problem, therefore optimal solution can take a long time to compute

How is graph coloring related to register allocation?

Chaitin's Graph Coloring Theorem

- Key observation: If a graph G has a node X with degree less than *n* (i.e. having less than *n* edges connected to it), then G is *n*-colorable *IFF* the reduced graph G' obtained from G by deleting X and all its edges is *n*-colorable.

Proof:



Graph Coloring Algorithm *(Not Optimal)*

- Assume the register interference graph is *n*-colorable

  How do you choose n?

- Simplification
  - Remove all nodes with degree less than *n*
  - Repeat until the graph has *n* nodes left
- Assign each node a different color
- Add removed nodes back one-by-one and pick a legal color as each one is added (2 nodes connected by an edge get different colors)

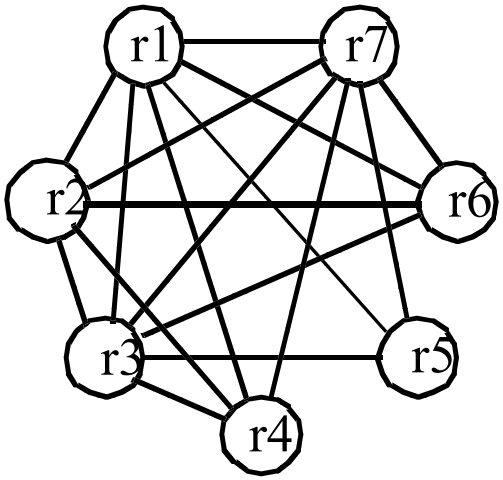
  Must be possible with less than n colors

- *Complications:* simplification can block if there are no nodes with less than *n* edges

  Choose one node to spill based on spilling heuristic

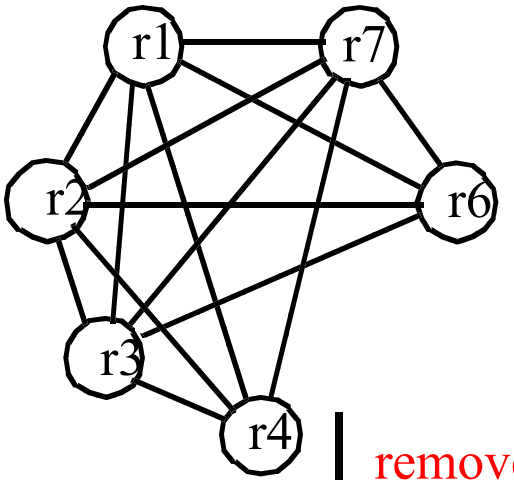
Example (N = 5)

COLOR stack = {}

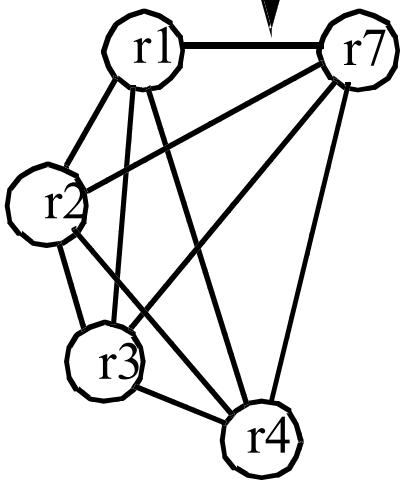


remove r5

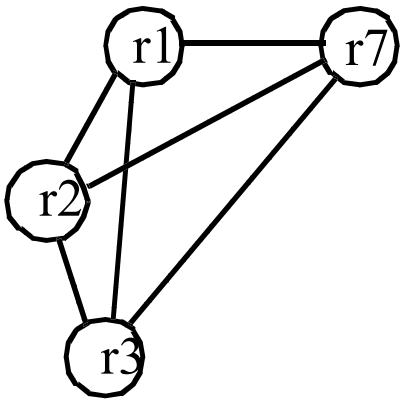
COLOR stack = {r5}



remove r6



remove r4

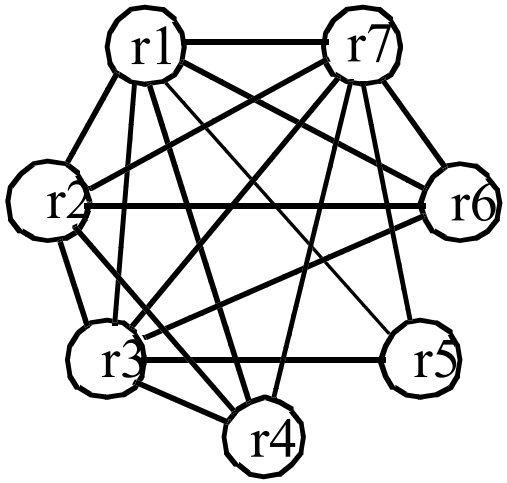


COLOR stack = {r5, r6, r4}

COLOR stack = {r5, r6}

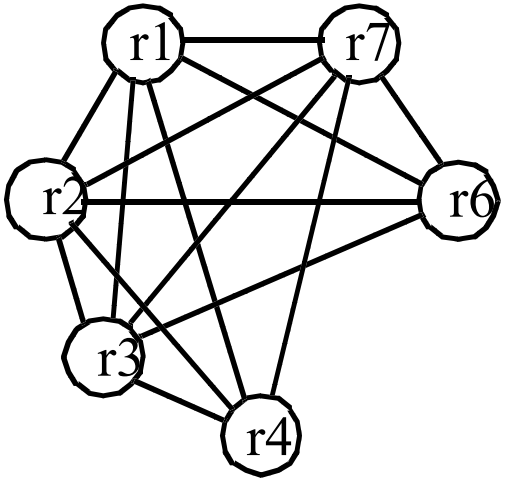
Example (N = 4)

COLOR stack = {}



remove r5

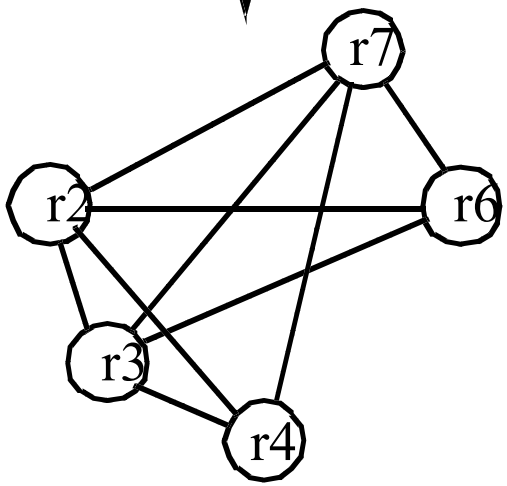
COLOR stack = {r5}



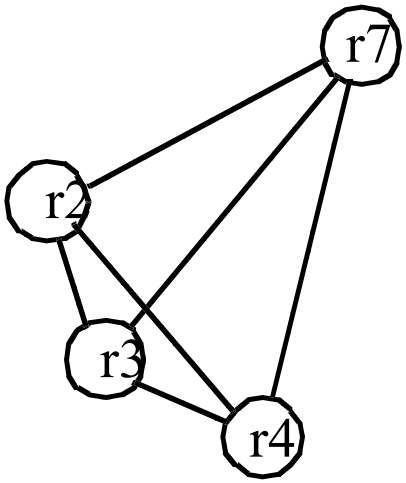
blocks

spill r1

Is this a ood choice??



remove r6



COLOR stack = {r5, r6}

COLOR stack = {r5}

Register Spilling

- When simplification is blocked, pick a node to delete from the graph in order to unblock
- Deleting a node implies the variable it represents will not be kept in register (i.e. spilled into memory)
  - When constructing the interference graph, each node is assigned a value indicating the estimated cost to spill it.
  - The estimated cost can be a function of the total number of definitions and uses of that variable weighted by its estimated execution frequency.
  - When the coloring procedure is blocked, the node with the least spilling cost is picked for spilling.
- When a node is spilled, spill code is added into the original code to store a spilled variable at its definition and to reload it at each of its use
- After spill code is added, a new interference graph is rebuilt from the modified code, and n-coloring of this graph is again attempted

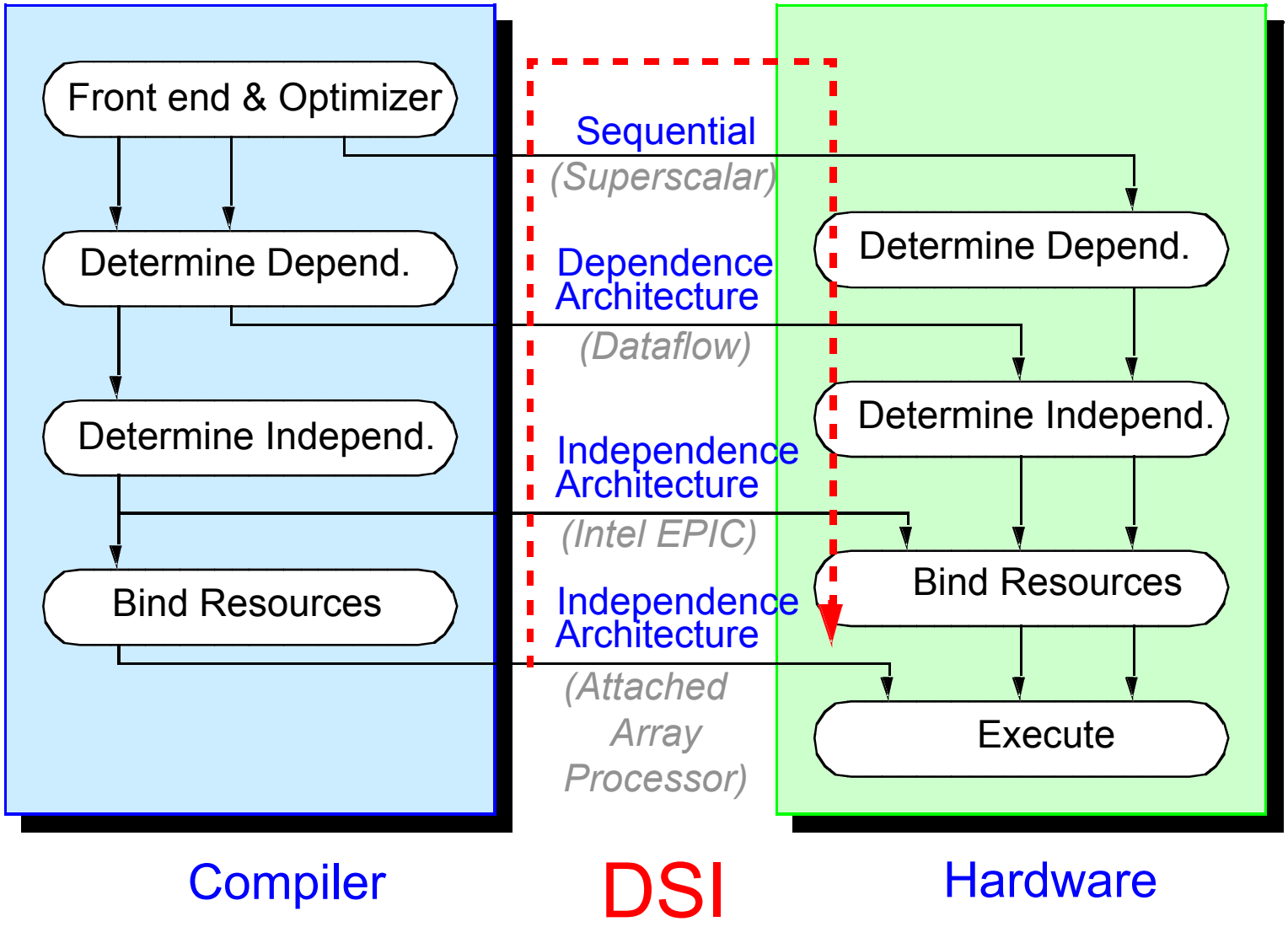
Phase Ordering Problem

- Register allocation prior to code scheduling
  - false dependencies induced due to register reuse
  - anti and output dependencies impose unnecessary constraints
  - code motion unnecessarily limited

- Code scheduling prior to register allocation
  - increase date live time (between creation and consumption)
  - overlap otherwise disjoint live ranges (increase register pressure)
  - may cause more live ranges to spill (run out of registers)
  - spill code produced will not have been scheduled

One option: do both prepass and postpass scheduling.

Compiler/Hardware Interactions



[B. Rau & J. Fisher, 1993]