Register Allocation

Part of the compiler’s back end

Critical properties
• Produce correct code that uses \( k \) (or fewer) registers
• Minimize added loads and stores
• Minimize space used to hold spilled values
• Operate efficiently
  \( O(n), O(n \log_2 n) \), maybe \( O(n^2) \), but not \( O(2^n) \)

Global Register Allocation

The big picture

At each point in the code
1. Determine which values will reside in registers
2. Select a register for each such value
   
The goal is an allocation that “minimizes” running time

Most modern, global allocators use a graph-coloring paradigm
• Build a “conflict graph” or “interference graph”
• Find a \( k \)-coloring for the graph, or change the code to a nearby problem that it can \( k \)-color
Global Register Allocation

What's harder across multiple blocks?
- Could replace a load with a move
- Good assignment would obviate the move
- Must build a control-flow graph to understand inter-block flow
- Can spend an inordinate amount of time adjusting the allocation

A more complex scenario
- Block with multiple predecessors in the control-flow graph
- Must get the “right” values in the “right” registers in each predecessor
- In a loop, a block can be its own predecessors
This adds tremendous complications
Global Register Allocation

Taking a global approach
- Abandon the distinction between local & global
- Make systematic use of registers or memory
- Adopt a general scheme to approximate a good allocation

Graph coloring paradigm (Lavrov & (later) Chaitin)

1. Build an interference graph $G_i$ for the procedure
   - Computing LIVE is harder than in the local case
   - $G_i$ is not an interval graph
2. (try to) construct a $k$-coloring
   - Minimal coloring is NP-Complete
   - Spill placement becomes a critical issue
3. Map colors onto physical registers

Graph Coloring (A Background Digression)

The problem

A graph $G$ is said to be $k$-colorable iff the nodes can be labeled with integers $1\ldots k$ so that no edge in $G$ connects two nodes with the same label

Examples

2-colorable

3-colorable

Each color can be mapped to a distinct physical register
Building the Interference Graph

What is an “interference”? (or conflict)

- Two values interfere if there exists an operation where both are simultaneously live
- If \( x \) and \( y \) interfere, they cannot occupy the same register

To compute interferences, we must know where values are “live”

The interference graph, \( G_i \)

- Nodes in \( G_i \) represent values, or live ranges
- Edges in \( G_i \) represent individual interferences
  > For \( x, y \in G_i \), \( \langle x, y \rangle \in E \) if \( x \) and \( y \) interfere
- A \( k \)-coloring of \( G_i \) can be mapped into an allocation to \( k \) registers

Building the Interference Graph

To build the interference graph

1. Discover live ranges
   > Build SSA form
   > At each \( \phi \)-function, take the union of the arguments

2. Compute LIVE sets for each block
   > Use an iterative data-flow solver
   > Solve equations for LIVE over domain of live range names

3. Iterate over each block
   > Track the current LIVE set
   > At each operation, add appropriate edges & update LIVE
   - Edge from result to each value in LIVE
   - Remove result from LIVE
   - Edge from each operand to each value in LIVE
What is a Live Range?

- A set LR of definitions \( \{d_1, d_2, \ldots, d_n\} \) such that for any two definitions \( d_i \) and \( d_j \) in LR, there exists some use \( u \) that is reached by both \( d_i \) and \( d_j \).
- How can we compute live ranges?
  > For each basic block \( b \) in the program, compute \( \text{REACHESOUT}(b) \) — the set of definitions that reach the exit of basic block \( b \)
    \[ \rightarrow d \in \text{REACHESOUT}(b) \text{ if there is no other definition on some path from } d \text{ to the end of block } b \]
  > For each basic block \( b \), compute \( \text{LIVEIN}(b) \) — the set of variables that are live on entry to \( b \)
    \[ \rightarrow v \in \text{LIVEIN}(b) \text{ if there is a path from the entry of } b \text{ to a use of } v \text{ that contains no definition of } v \]
  > At any block where control flow joins, for each live variable \( v \), merge the live ranges associated with definitions in \( \text{REACHESOUT}(p) \), for all predecessors of \( b \), that assign a value to \( v \).

Computing LIVE Sets

A value \( v \) is live at \( p \) if \( \exists \) a path from \( p \) to some use of \( v \) along which \( v \) is not re-defined

Data-flow problems are expressed as simultaneous equations

\[
\begin{align*}
\text{LIVEOUT}(b) &= \bigcup_{s \in \text{succ}(b)} \text{LIVEIN}(s) \\
\text{LIVEIN}(b) &= (\text{LIVEOUT}(b) \cap \text{NOTDEF}(b)) \cup \text{IN}(b)
\end{align*}
\]

where

- \( \text{IN}(x) \) is the set of names used before redefinition in block \( x \)
- \( \text{NOTDEF}(x) \) is the set of names not redefined in \( x \)

As output,

- \( \text{LIVEOUT}(x) \) is the set of names live on exit from block \( x \)
- \( \text{LIVEIN}(x) \) is the set of names live on entry to block \( x \)
Computing LIVE Sets

The compiler solves the equations with an iterative algorithm.

\[
\text{WorkList} \leftarrow \{ \text{all blocks} \}
\]
while (WorkList \neq \emptyset)
remove a block b from WorkList
Compute LIVEOUT(b) Compute LIVEIN(b)
if LIVEIN(b) changed
then add \text{pred}(b) to WorkList

Why does this work?
• \text{LIVEOUT}, \text{LIVEIN} \subseteq \mathcal{P}(\text{Name})
• \text{IN}, \text{NOTDEF} are constant for b
• Equations are monotone
• Finite chains in the lattice
⇒ will reach a fixed point!

Speed of convergence depends on the order in which blocks are “removed” & their sets recomputed.

This is the world’s quickest introduction to data-flow analysis!

Observation on Coloring for Register Allocation

- Suppose you have \( k \) registers—try to color the graph with \( k \) colors.
- Any vertex \( n \) that has fewer than \( k \) neighbors in the interference graph \((\text{\(n^{\circ} < k\)})\) can always be colored!
  - Pick any color not used by its neighbors — there must be one.
- Idea for Chaitin’s algorithm:
  - Pick any vertex \( n \) such that \( n^{\circ} < k \) and put it on the stack.
  - Remove that vertex and all edges incident from the interference graph.
    → This may make some new nodes have fewer than \( k \) neighbors.
  - At the end, if some vertex \( n \) still has \( k \) or more neighbors, then spill the live range associated with \( n \).
  - Otherwise successively pop vertices off the stack and color them in the lowest color not used by some neighbor.
**Chaitin’s Algorithm**

1. While there are vertices with fewer than \( k \) neighbors remaining in the interference graph \( G_i \):
   - Pick any vertex \( n \) such that \( n^i < k \) and put it on the stack
   - Remove that vertex and all edges incident to it from \( G_i \)
     - This may cause additional vertices to have fewer than \( k \) neighbors

2. If any vertices remain in the interference graph \( G_i \) (all such vertices have \( k \) or more neighbors) then:
   - Pick a vertex \( n \) (using some heuristic condition) and spill the live range associated with \( n \)
   - Remove vertex \( n \) from \( G_i \), along with all edges incident to it and put it on the stack
   - If this causes some vertex in \( G_i \) to have fewer than \( k \) neighbors, then go to step 1; otherwise, repeat step 2

3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor

---

**Chaitin’s Algorithm in Practice**

3 Registers

Stack

- 1
- 2
- 3
- 4
- 5
Chaitin’s Algorithm in Practice

3 Registers

Stack

Chaitin’s Algorithm in Practice

3 Registers

Stack
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 

CMSC430 Spring 2007
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 

Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: ○
2: ●
3: □

Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: ○
2: ●
3: □
Chaitin’s Algorithm in Practice

3 Registers

Colors:
1: \[ \text{Color 1} \]
2: \[ \text{Color 2} \]
3: \[ \text{Color 3} \]

Improvement in Coloring Scheme

- Due to Briggs, Cooper, Kennedy, and Torczon
- Instead of stopping at the end when all vertices have at least \( k \) neighbors, put each on the stack according to some priority
  > When you pop them off they may still color!

2-Colorable

2 Registers:
**Chaitin-Briggs Algorithm**

1. While there are vertices with fewer than \( k \) neighbors remaining in the interference graph \( G_i \):
   - Pick any vertex \( n \) such that \( n^i < k \) and put it on the stack.
   - Remove that vertex and all edges incident to it from \( G_i \).
     - This may create vertices with fewer than \( k \) neighbors.

2. If any vertices remain in the interference graph \( G_i \) (all such vertices have \( k \) or more neighbors) then:
   - Pick a vertex \( n \) (using some heuristic condition), put it on the stack and remove vertex \( n \) from \( G_i \), along with all edges incident to it.
   - If this causes some vertex in \( G_i \) to have fewer than \( k \) neighbors, then go to step 1; otherwise, repeat step 2.

3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor:
   - If some vertex cannot be colored, then pick a live range to spill, spill it, and restart at step 1.

---

**Chaitin Allocator**  
*(Bottom-up Coloring)*

- **renumber**  
- **build**  
- **coalesce**  
- **spill costs**  
- **simplify**  
- **select**  
- **spill**

Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies

\[ LR_x \rightarrow LR_y, \text{ and } < LR_x, LR_y > \notin G_i \Rightarrow \text{combine } LR_x \& LR_y \]

Estimate cost for spilling each live range

While stack is non-empty:

- **pop** \( n \), insert \( n \) into \( G_i \), & try to color it

- **spill** uncolored definitions & uses

While stack is non-empty:

- **pop** \( n \), insert \( n \) into \( G_i \), & try to color it

- **spill** uncolored definitions & uses

while \( N \) is non-empty:

- **push** \( n \) onto stack

- **remove** \( n \) from \( G_i \)
**Chaitin-Briggs Allocator** *(Bottom-up Coloring)*

1. **renumber**
   - Build SSA, build live ranges, rename

2. **build**
   - Build the interference graph

3. **coalesce**
   - Fold unneeded copies
   
   \[ \text{LR}_x \rightarrow \text{LR}_y, \text{and } < \text{LR}_x, \text{LR}_y> \notin G \Rightarrow \text{combine LR}_x \text{ & LR}_y \]

4. **spill costs**
   - Estimate cost for spilling each live range

5. **simplify**
   - Remove nodes from the graph

6. **select**
   - While stack is non-empty
     - pop n, insert n into GI, & try to color it
     - Spill uncolored definitions & uses

7. **spill**

---

**Briggs’ algorithm** *(1989)*

---

**Picking a Spill Candidate**

When \( \exists n \in G_n, n \geq k \), simplify must pick a spill candidate

Chaitin’s heuristic

- Minimize spill cost + current degree
- If LR has a negative spill cost, spill it pre-emptively
  > Cheaper to spill it than to keep it in a register
- If LR has an infinite spill cost, it cannot be spilled
  > No value dies between its definition & its use

Spill cost is weighted cost of loads & stores needed to spill x

Bernstein *et al.* Suggest repeating simplify, select, & spill with several different spill choice heuristics & keeping the best
**Other Improvements to Chaitin-Briggs**

**Spilling partial live ranges**
- Bergner introduced interference region spilling
- Limits spilling to regions of high demand for registers

**Splitting live ranges**
- Simple idea — break up one or more live ranges
- Lets allocator use different registers for distinct subranges
- Lets allocator spill subranges independently \(\text{(use 1 spill location)}\)

**Conservative coalescing**
- Combining \(L_{Rx} \rightarrow L_{Ry}\) to form \(L_{Rxy}\) may increase register pressure
- Limit coalescing to case where \(L_{Rxy} < k\)
- Iterative form tries to coalesce before spilling

---

**Chaitin-Briggs Allocator** (Bottom-up Global)

**Strengths & weaknesses**
- \(\uparrow\) Precise interference graph
- \(\uparrow\) Strong coalescing mechanism
- \(\uparrow\) Handles register assignment well
- \(\uparrow\) Runs fairly quickly
- \(\downarrow\) Known to overspill in tight cases
- \(\downarrow\) Interference graph has no geography
- \(\downarrow\) Spills a live range everywhere
- \(\downarrow\) Long blocks devolve into spilling by use counts

Is improvement possible?  
- With rising spill costs, aggressive transformations, & long blocks
What about Top-down Coloring?

- The Big Picture
  - Use high-level priorities to rank live ranges
  - Allocate registers for them in priority order
  - Use coloring to assign specific registers to live ranges

- The Details
  - Separate constrained from unconstrained live ranges
    - A live range is constrained if it has \( \geq k \) neighbors in \( G \)
  - Color constrained live ranges first
  - Reserve pool of local registers for spilling (or spill & iterate)
  - Chow split live ranges before spilling them
    - Split into block-sized pieces
    - Recombine as long as \( ^{\leq} k \)

Tradeoffs in Global Allocator Design

- Top-down versus bottom-up
  - Top-down uses high-level information
  - Bottom-up uses low-level structural information

- Spilling
  - Reserve registers versus iterative coloring

- Precise versus imprecise graph
  - Precision allows coalescing
  - Imprecision speeds up graph construction

**Big-iron** ⇒ precise, iterated, bottom-up
**JIT** ⇒ imprecise, reserve, b-u or t-d
Regional Approaches to Allocation

Hierarchical Register Allocation  (Koblenz & Callahan)

• Analyze control-flow graph to find hierarchy of tiles
• Perform allocation on individual tiles, innermost to outermost
• Use summary of tile to allocate surrounding tile
• Insert compensation code at tile boundaries (LR_x→LR_y)

Strengths
→ Decisions are largely local
→ Use specialized methods on individual tiles
→ Allocator runs in parallel

Weaknesses
→ Decisions are made on local information
→ May insert too many copies
Still, a promising idea

• Anecdotes suggest it is fairly effective
• Target machine is multi-threaded multiprocessor  (Tera MTA)

Proportional Register Allocation  (Proebsting & Fischer)

• Attempt to generalize from Best’s algorithm (bottom-up, local)
• Generalizes “furthest next use” to a probability
• Perform an initial local allocation using estimated probabilities
• Follow this with a global phase
  > Compute a merit score for each LR as (benefit from x in a register = probability it stays in a register)
  > Allocate registers to LRs in priority order, by merit score, working from inner loops to outer loops
  > Use coloring to perform assignment among allocated LRs

• Little direct experience (either anecdotal or experimental)
• Combines top-down global with bottom-up local
Regional Approaches to Allocation

Register Allocation via Fusion (Lueh, Adl-Tabatabi, Gross)

- Use regional information to drive global allocation
- Partition CFGs into regions & build interference graphs
- Ensure that each region is k-colorable
- Merge regions by fusing them along CFG edges
  - Maintain k-colorability by splitting along fused edge
  - Fuse in priority order computed during the graph partition
- Assign registers using int. graphs
  \( i.e., \) execution frequency

**Strengths**
- Flexibility
- Fusion operator splits on low-frequency edges

**Weaknesses**
- Choice of regions is critical
- Breaks down if region connections have many live values

---

List Scheduling

\[
\begin{align*}
\text{Cycle} & \leftarrow 1 \\
\text{Ready} & \leftarrow \text{leaves of } P \\
\text{Active} & \leftarrow \emptyset \\
\text{while} \ (\text{Ready} \cup \text{Active} \neq \emptyset) & \\
\text{if} \ (\text{Ready} \neq \emptyset) & \text{then} \\
& \text{remove an op from Ready} \\
& \text{S(op)} \leftarrow \text{Cycle} \\
& \text{Active} \leftarrow \text{Active} \cup \text{op} \\
\text{Cycle} & \leftarrow \text{Cycle} + 1 \\
\text{for each op} & \in \text{Active} \\
\text{if} \ (\text{S(op)} + \text{delay(op)} \leq \text{Cycle}) & \text{then} \\
& \text{remove op from Active} \\
\text{for each successor s of op in P} & \\
\text{if} \ (s \text{ is ready}) & \text{then} \\
& \text{Ready} \leftarrow \text{Ready} \cup s
\end{align*}
\]

**Removal in priority order**

**Note:** only one op per cycle

**Op has completed execution**

**If successor’s operands are ready, put it on Ready**
**Detailed Scheduling Algorithm I**

**Idea:** Keep a collection of worklists $W[c]$, one per cycle

> We need $MaxC = \text{max delay} + 1$ such worklists

**Code:**

```plaintext
for each $n \in \mathbb{N}$ do begin
    count[$n$] := 0; earliest[$n$] = 0 end
for each ($n_1,n_2) \in \mathbb{E}$ do begin
    count[$n_2$] := count[$n_2$] + 1;
    successors[$n_1$] := successors[$n_1$] $\cup$ {n2};
end
for i := 0 to $\text{MaxC} - 1$ do
    $W[i]$ := $\emptyset$;
    $Wcount$ := 0;
for each $n \in \mathbb{N}$ do
    if count[$n$] = 0 then begin
        $W[0]$ := $W[0]$ $\cup$ {n}; $Wcount$ := $Wcount$ + 1;
    end
$c := 0$; // $c$ is the cycle number
$cW := 0$; // $cW$ is the number of the worklist for cycle $c$
instr[$c$] := $\emptyset$;

Idea: Keep a collection of worklists $W[c]$, one per cycle

We need $MaxC = \text{max delay} + 1$ such worklists

**Detailed Scheduling Algorithm II**

```plaintext
while $Wcount > 0$ do begin
    while $W[cW] = \emptyset$ do begin
        $c := c + 1$; instr[$c$] := $\emptyset$; $cW := \text{mod}(cW+1,MaxC)$;
    end
    nextc := $\text{mod}(c+1,MaxC)$;
    while $W[cW] \neq \emptyset$ do begin
        select and remove an arbitrary instruction $x$ from $W[cW]$;
        if $\exists$ free issue units of type($x$) on cycle $c$ then begin
            instr[$c$] := instr[$c$] $\cup$ {x}; $Wcount$ := $Wcount$ - 1;
            for each $y \in \text{successors}[x]$ do begin
                count[$y$] := count[$y$] - 1;
                earliest[$y$] := max(earliest[$y$], $c+$delay($x$));
                if count[$y$] = 0 then begin
                    loc := $\text{mod}(\text{earliest}[y],MaxC)$;
                    $W[loc]$ := $W[loc]$ $\cup$ {y}; $Wcount$ := $Wcount$ + 1;
                end
            end
        end
        else $W[nextc] := W[nextc] \cup \{x\}$;
    end
end
```

**Priority**
**Instruction Scheduling** *(revisited)*

In an earlier lecture, we introduced list scheduling

- Efficient, greedy, local heuristic
- Technique of choice for more than 20 years

How can the compiler improve on local list scheduling?

- Different priority functions & tie breakers
- Use forward & backward list scheduling (Figures 12.4 & 12.5)
- Increase size of region fed to scheduler *(classic answer)*
- Try other algorithms

Little success with other algorithms on code written by humans

- Some compiler generated code defeats the list scheduler

---

**What About OOO Execution?**

Out-of-order microprocessors should simplify scheduling

- Processor looks at a window in the instruction stream
- Processor executes operations as they are ready
- Processor (typically) renames registers for correctness

Does this eliminate the need for instruction scheduling?

- For any finite window, ∃ a worst case schedule
  - 100 operation window ⇒ 110 for FU₁, 110 for FU₂
- Schedule need not be perfect, but must be not bad
- OOO can compensate for mild variations in latency
  - Cache miss, infrequent stall, ...

Depth-first schedule!
Other Priority Functions

Computing Ranks

- Maximum path length containing it
  - Favors critical path; tends towards depth-first
- Number of immediate successors in P
  - Favors longer ready queue; tends toward breadth-first
- Total number of descendants in P
  - Favors heavily used values; tends toward breadth-first
- Add latency to node’s rank
  - Favors long operations to cover their latencies
- Increment rank of if node contains a last use
  - Tends to shorten live ranges & decrease register pressure

These can be used as priorities or as tie-breakers
  - Use randomization & repetition

Forward versus Backward List Scheduling

Folk wisdom has long suggested doing both

- Some blocks amenable to forward scheduling
- Some blocks amenable to backward scheduling
- Conventional approach is to try both & keep best result

Does it matter?

- Takes dependence graph that is both wide & deep
- Depends on detailed knowledge of the specific block
- See Figures 12.4 & 12.5 in EAC for a real example

Advice

- Use several forward & several backward passes
- Use different priorities & tie breakers, or use randomization
How Well Does List Scheduling Do?

Non-optimal list schedules (%) versus available parallelism
1 functional unit, randomly generated blocks of 10, 20, 50 ops

At the peak, compiler should apply other techniques
- Measure parallelism in list scheduler
- Invoke stronger techniques when high-probability of payoff

Scheduling over Larger Scopes

Basic idea is simple
- Longer sequence of operations ⇒ more opportunities
- Pick a multi-block path & treat it as a single block
- Add compensation code for other exits (& entries)

Several distinct scopes
- Extended basic blocks
  - A sequence $b_1, b_2, ..., b_n$ where $b_i$ has 1 predecessor, $1 < i \leq n$
- Traces
  - An arbitrary acyclic path, usually chosen from trace data
- Loops
  - Think of source-language loop, can find arbitrary loops
**Extended Basic Blocks**

An example

Extended Basic Blocks

→ b₁, b₂, b₄
→ b₁, b₃
→ b₅
→ b₆
Both b₅ and b₆ have > 1 predecessors

Scheduling EBBs
- Treat EBB as a block
- Moving an operation across boundary can necessitate compensation code
- Can restrict motion to zero growth case with 12 to 13% improvement [LCTES 98]

**Traces**

Technique developed for the Multiflow Computer

1 Identify high-frequency path
2 Schedule it as if a single block
3 Insert compensation code
4 Schedule next important path

Results
- Fast trip through common path
- Off-path quality declines
- Compensation code \(\Rightarrow\) growth
Loop Scheduling (Software Pipelining)

Loops deserve special attention

- Their bodies execute frequently
- They do most of the work in time-critical computations
- They often contain major holes & interlocks

The ideas

- Schedule multiple iterations together
- Run several iterations concurrently
- Shorten “initiation interval” for overall speed
  - Cycles between initiation of different iterations
  - $= \text{length of computation kernel}$

Example

```
loadI r0,0 ⇒ r1
loadI r0,400 ⇒ r2
floatAI r0,c ⇒ fr1
10 floatAI r1,a ⇒ fr2
11 fadd fr2,fr1 ⇒ fr2
12 floatAE fr2 ⇒ r1,b
13 addI r1,8 ⇒ r1
14 cmp_LE r1,r2 ⇒ r3
15 chr r1 ⇒ 10,16
```

2 cycle delay

3 cycle delay

How fast (in cycles per iteration) can we execute this loop?
Minimum Number of Cycles in Kernel

• Machine resource constraint:
  - \( N_u \) is the number of units of type \( u \)
  - \( I_u \) is the number of instructions requiring a unit of type \( u \)
  - \( \left\lfloor \frac{I_u}{N_u} \right\rfloor \) is the minimum number of cycles required for one iteration of the loop based on unit \( u \)
  - \( \max_u \left\lfloor \frac{I_u}{N_u} \right\rfloor \) is the minimum number of cycles required for all units

• Slope constraint
  - If the loop computes a recurrence over \( k_r \) cycles
  - And the total delay along the recurrence cycle is \( d_r \)
  - Then each iteration is going to require \( \frac{d_r}{k_r} \) cycles to execute
  - \( \max_r \left\lfloor \frac{d_r}{k_r} \right\rfloor \) is the minimum number of cycles per iteration to compute recurrences

Example

```plaintext
loadI r0,0 ⇒ r1
loadI r0,400 ⇒ r2
floadAI r0,c ⇒ fr1
10 floadAI r1,a ⇒ fr2
11 fadd fr2,fr1 ⇒ fr2
12 fistoreAI fr2 ⇒ r1,b
13 addI r1,8 ⇒ r1
14 cmp_LT r1, r2 ⇒ r3
15 chr r3 ⇒ 10,16
```

Floating Pt Unit: 1 instruction
Load/Store Unit: 2 instructions
Integer Unit: 3 instructions

⇒ 3 cycles minimum
Loop Scheduling

Mechanics

• Determine lower bound on initiation interval
  > Number of issue slots
  > Longest dependence chain
  \[\text{Register constraint, too}\]

• Lay out a schedule of appropriate length

• Use list scheduling with a modulo cycle count
  > Could fail — just try with one more cycle per iteration

• Add a pre-loop & a post-loop to “fill” & “drain” the pipeline

It gets pretty intricate!

• Conditional control flow complicates matters even more

Example

```
loadI r0, 0 => r1
loadI r0, 400 => r2
floatAI r0,c => fr1
10  floatAI r1,a => fr2
11  fadd  fr2,fr1 => fr2
12  fstoreAI fr2 => r1,b
13  addI  r1,8 => r1
14  cmp_LE r1,r2 => r3
15  cbrr r1 => l0,l6
```

Floating Point Unit
Integer Unit
Load/Store Unit

<table>
<thead>
<tr>
<th>Load/Store Unit</th>
<th>Integer Unit</th>
<th>Floating Point Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>floatAI r1,a =&gt; fr2</td>
<td>addI r1,8 =&gt; r1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cmp LE r1,r2 =&gt; r3</td>
<td></td>
</tr>
<tr>
<td>fstoreAI fr3 =&gt; r1,b-16</td>
<td>cbrr r3 =&gt; 10,16</td>
<td>fadd fr2,fr1 =&gt; fr3</td>
</tr>
</tbody>
</table>

2 cycle delay
3 cycle delay
**Final Code**

    ld  r1,0
    ld  r2,400
    flid fr1, c
p1    flid fr2,a(r1);   ai  r1,r1,8
p2    comp r1,r2
p3    beq  e1;    fadd  fr3,fr2,fr1
k1    flid fr2,a(r1);   ai  r1,r1,8
k2    comp r1,r2
k3    fst fr3,b-16(r1); blek1;   fadd  fr3,fr2,fr1
e1    nop
e2    nop
e3    fst fr3,b-8(r1)