Lab 3, Tutorial 1

Comp 412

Copyright 2018, Keith D. Cooper, Linda Torczon & Zoran Budimlić, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.
## Lab 3 Schedule

<table>
<thead>
<tr>
<th>Sunday</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
<th>Saturday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

- **You are here**
- **The goal is here**
- **Lab 3 Due Date**
- **Thanksgiving Break**
- **Docs Available Lab Lecture**
- **Start work**
- **Correct Schedule**
- **Improve Schedules**

**Lab 3 Due Date**
- Late day

**Thanksgiving Break**
- Not a late day
- Not a late day
- Not a late day
Lab 3 Builds On Lab 2

There are a number of similarities between lab 2 & lab 3

• Same set of ILOC operations
  – With some new twists
  – And, nop has a purpose

• Same set of tools
  – Lab 3 Reference Implementation
  – Lab 3 ILOC Simulator
    → It seems obvious, but you must use the lab 3 simulator, not the lab 2 simulator

• Same kind of testing scripts
  – Make a copy, and change the defined variable SCHED in SchedHelper
  – By default, the scripts run lab3_ref

• A large library of ILOC test blocks

Poke around in ~/comp412/students/lab3 on CLEAR
Lab 3 Differs From Lab 2

Lab 3 tools are different than the lab 2 tools

• **ILOC** now has a notation for starting two ops in a single cycle
  – \([ op_1 ; op_2 ]\) tells the simulator to start \(op_1\) and \(op_2\) in the same cycle
    → They **start** at the same time; they **finish** when they are done
    → **ILOC** used as input to your scheduler has only one operation per instruction
  – At most 1 **load** or **store**, and at most 1 **mult** in a given instruction
  – Simulator **dispatches** the operations to the appropriate functional unit

• Simulator has a flag that sets the interlock mode
  – Use `-s 3` to determine number of cycles for original code
    → Tells the simulator to enforce all interlocks (& is the default setting)
  – Use `--s 1` when testing your scheduler’s effectiveness & output code’s speed
    → Tells the simulator to ignore interlocks based on values (& must be specified)
Lab 3: The Big Picture

The key steps in your scheduler should be: (in order)

1. Scan, parse, & rename the input ILOC code
2. Build a dependence graph for the block being scheduled
3. Compute priorities for each operation
4. Schedule the code in a way that preserves the dependences

Algorithms

• You implemented the front end for the lab 2 code check
  – Best practice: reuse that code (& wish you had commented it)
• Dependence Graph: see lecture notes from lab 3 primer (e.g., L27, slide 14)
• Priorities: lots of variation possible. (see Gibbons & Muchnick, or slide 17)
  – Common schemes: latency-weighted depth, breadth first, depth first
• Scheduler: most people build greedy list schedulers (e.g., slide 19)

Read § 12.3 in EaC2e
Scary Admonition

**Warning:** Your scheduler must not perform optimizations other than instruction scheduling and register renaming. It must not remove useless (dead) instructions; it must not fold constants. Any instruction left in the code by the optimizer is presumed to have a purpose. Assume that the optimizer had a firm basis for not performing the optimization. Your lab should schedule the instructions, not search for other opportunities for improvement. *This lab is an investigation of instruction scheduling, not a project in general optimization algorithms.*

Additionally, your scheduler must not rely on input constants specified in comments. When the TAs grade your scheduler, they are not required to use the input specified in the comments. Your scheduler is expected to generate correct code regardless of whether or not the TAs use the input specified in the comments.

If the result of an op is not used, you cannot delete the op.
For operations other than load, store, & output, dependence is straightforward

- Each use is has an edge to the operation that defines it
  - Definition is the edge’s sink, use is the edge’s source

- Each edge has a latency equal to that of its sink operation

1. loadI 8 => r1
2. loadI 12 => r2
3. mult r1, r2 => r3
4. add r1, r3 => r4

Original Code

```
[loadI 12 => r3 ;
loadI 8 => r1]
[nop ;
multI r1,r3 => r2]
[nop ;
nop]
[nop ;
nop]
[add r1, r2 => r4 ;
nop ]
```

Dependence Graph

Scheduled Code (lab3_ref)

Credits: Schedule and graph produced by “lab3_ref –s –g” on “original code.” Drawing by graphviz.
Dependence Graph Construction

Create an empty map, $M$
walk the block, top to bottom
at each operation $o$:
create a node for $o$
if it defines $VR_i$:
set $M(\text{VR}_i)$ to $o$
for each $\text{VR}_j$ used in $o$, add an edge from $o$ to the node in $M(\text{VR}_j)$
if $o$ is a load, store, or output operation, add edges to ensure serialization of memory ops

Building the graph

1. $M[*] \leftarrow \bot$

2. Create node for op 1
   Set $M(r1)$ to node 1

Example

1.\hspace{10pt}loadl 8 => r1
2.\hspace{10pt}loadl 12 => r2
3.\hspace{10pt}mult r1, r2 => r3
4.\hspace{10pt}add r1,r3 => r4
Dependence Graph Construction

Create an empty map, $M$
walk the block, top to bottom
   at each operation $o$:
      create a node for $o$
         if it defines $VR_j$
            set $M(VR_j)$ to $o$
      for each $VR_j$ used in $o$, add an edge
         from $o$ to the node in $M(VR_j)$
      if $o$ is a load, store, or output
         operation, add edges to ensure
         serialization of memory ops

Building the graph
1. $M[*] \leftarrow \perp$
2. Create node for op 1
   Set $M(r_1)$ to node 1
3. Create node for op 2
   Set $M(r_2)$ to node 2

Example
1 loadI 8 => r1
2 loadI 12 => r2
3 mult r1, r2 => r3
4 add r1,r3 => r4
Dependence Graph Construction

Create an empty map, $M$
walk the block, top to bottom
at each operation $o$:
create a node for $o$
if it defines $VR_i$:
set $M(VR_i)$ to $o$
for each $VR_j$ used in $o$, add an edge from $o$ to the node in $M(VR_j)$
if $o$ is a load, store, or output operation, add edges to ensure serialization of memory ops

Building the graph
1. $M[*] \leftarrow \bot$
2. Create node for op 1
   Set $M(r1)$ to node 1
3. Create node for op 2
   Set $M(r2)$ to node 2
4. Create node for op 3
   Set $M(r3)$ to node 3

Example
1. loadI 8 => r1
2. loadI 12 => r2
3. mult r1, r2 => r3
4. add r1,r3 => r4

COMP 412, Fall 2018
Dependence Graph Construction

Create an empty map, $M$
walk the block, top to bottom
  at each operation $o$:
    create a node for $o$
    if it defines $\mathbf{VR}_i$:
      set $M(\mathbf{VR}_i)$ to $o$
    for each $\mathbf{VR}_j$ used in $o$, add an edge from $o$ to the node in $M(\mathbf{VR}_j)$
    if $o$ is a load, store, or output operation, add edges to ensure serialization of memory ops

Building the graph

1. $M[*] \leftarrow \perp$

2. Create node for op 1
   Set $M(r1)$ to node 1

3. Create node for op 2
   Set $M(r2)$ to node 2

4. Create node for op 3
   Set $M(r3)$ to node 3
   Op 3 uses r1 & r2, so add edges to $M(r1)$ and $M(r2)$

Example

1. loadl 8 => r1
2. loadl 12 => r2
3. mult r1, r2 => r3
4. add r1,r3 => r4

COMP 412, Fall 2018
Dependence Graph Construction

Create an empty map, $M$
walk the block, top to bottom
at each operation $o$:
  create a node for $o$
  if it defines $VR_i$:
    set $M(VR_i)$ to $o$
  for each $VR_j$ used in $o$, add an edge
    from $o$ to the node in $M(VR_j)$
if $o$ is a load, store, or output
  operation, add edges to ensure
  serialization of memory ops

Building the graph
1. $M[*] \leftarrow \bot$
2. Create node for op 1
   Set $M(r1)$ to node 1
3. Create node for op 2
   Set $M(r2)$ to node 2
4. Create node for op 3
   Set $M(r3)$ to node 3
   Op 3 uses r1 & r2, so add edges to $M(r1)$ and $M(r2)$
5. Create node for op 4
   Set $M(r4)$ to node 4

Example

1. loadI 8 => r1
2. loadI 12 => r2
3. mult r1, r2 => r3
4. add r1, r3 => r4
Dependence Graph Construction

Create an empty map, $M$

walk the block, top to bottom

at each operation $o$:

create a node for $o$

if it defines $VR_i$:

set $M(VR_i)$ to $o$

for each $VR_j$ used in $o$, add an edge from $o$ to the node in $M(VR_j)$

if $o$ is a load, store, or output operation, add edges to ensure serialization of memory ops

Example

1. loadl 8 => r1
2. loadl 12 => r2
3. mult r1, r2 => r3
4. add r1,r3 => r4

Building the graph

1. $M[*] \leftarrow \bot$
2. Create node for op 1
   Set $M(r1)$ to node 1
3. Create node for op 2
   Set $M(r2)$ to node 2
4. Create node for op 3
   Set $M(r3)$ to node 3
   Op 3 uses r1 & r2, so add edges to $M(r1)$ and $M(r2)$
5. Create node for op 4
   Set $M(r4)$ to node 4
   Op 4 uses r1 & r3, so add edges to $M(r1)$ and $M(r3)$
The Scheduler Needs Some Additional Edges

To ensure the correct flow of values, the scheduler needs edges that specify the relative ordering of loads, stores, and outputs.

<table>
<thead>
<tr>
<th>First op</th>
<th>Second op</th>
<th>Same address</th>
<th>Distinct addresses</th>
<th>Unknown address(es)</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>load</td>
<td>—</td>
<td>No conflict (after renaming)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>load or output</td>
<td>store</td>
<td>conflict</td>
<td>no conflict</td>
<td>conflict</td>
<td>WAR</td>
</tr>
<tr>
<td>store</td>
<td>store</td>
<td>conflict</td>
<td>no conflict</td>
<td>conflict</td>
<td>WAW</td>
</tr>
<tr>
<td>store</td>
<td>load or output</td>
<td>conflict</td>
<td>no conflict</td>
<td>conflict</td>
<td>RAW</td>
</tr>
<tr>
<td>output</td>
<td>output</td>
<td>need an edge to serialize the outputs</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

“conflict” implies 1\textsuperscript{st} op must complete before 2\textsuperscript{nd} op issues ⇒ an edge in dependence graph
Dependence

Dependence relations among memory operations are more complex

1. loadI 8 => r1
2. loadI 12 => r2
3. add r1, r2 => r3
4. load r3 => r4
5. load r1 => r5
6. store r4 => r3
7. output 12

Original Code

Dependence Graph

Scheduled Code (lab3_ref)
Dependence Graph Construction

Build a dependence graph to capture critical relationships in the code.

Explanatory Notes

1. ‘o’ refers to both the operation in the original block and the node that represents it in the graph. The meaning should be clear by context.

2. At a minimum, in the absence of other information:
   - load & output need an edge to the most recent store (full latency)
   - output needs an edge to the most recent output (serialization)
   - store needs an edge to the most recent store & output, as well as each previous load (serialization)

If your lab tries to simplify the graph, it may need more edges for store & output nodes.

From local scheduling lecture

Building the Graph

Create an empty map, $M$
walk the block, top to bottom
    at each operation $o$ that defines $VR_i$
        create a node for $o$\(^1\)
        set $M(VR_i)$ to $o$
    for each $VR_j$ used in $o$, add an edge from $o$ to the node in $M(VR_j)$
    if $o$ is a load, store, or output operation, add edges to ensure serialization of memory ops*
Dependence

Dependence relations among memory operations are more complex

1. loadl 8 => r1
2. loadl 12 => r2
3. add r1, r2 => r3
4. load r3 => r4
5. load r1 => r5
6. store r4 => r3
7. output 12

loadl 12 => r4
loadr 8 => r3
load r3 => r2
add r3, r4 => r0
load r0 => r1
nop
nop
top
top
store r1 => r0
top
top
top
top
output 12
nop

Why no IO edge here? It would be redundant.
Dependence Among Memory Operations

The key point is simple:

_The original code is the specification for the computation_

• The order of operations in the original code is correct
• Any legal schedule must produce equivalent results

Consequences

• Loads & outputs that precede a store in the original code must precede it in the scheduled code
  – _Unless_ the scheduler can _prove_ that the operations are disjoint
    → _Store never touches the same memory location as the load or output_

• Stores that precede a load or output in the original code must precede them in the scheduled code
  – _Unless_ the scheduler can _prove_ that the operations are disjoint

• Outputs must be strictly ordered, according to the original code
  – The order of execution of outputs _defines_ correctness
Dependence Among Memory Operations

The key point is simple:

*The original code is the specification for the computation*

- The order of operations in the original code is correct
- Any legal schedule must produce equivalent results

**Consequences**

- To preserve these relationships, you will add a lot of edges to the graph
  - Those edges constrain the schedule, of course ...
  - If they constrain the schedule needlessly, that hurts performance
- Many students implement some kind of graph simplification phase
  - Limits on what you can do (arbitrary, dictatorial limits)
  - Limits on what the analysis can detect

**First:** get the dependence graph *correct* & the scheduler *working*

**Second:** simplify the graph

Next week’s tutorial will look at graph simplification.
Representing the Graph

Appendix B.3 in EaC2e describes several graph representations

- `lab3_ref` uses the “tabular representation” on page 747
- It allows easy traversals in either direction
  - requires no actual pointers

### Node Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Succ</th>
<th>Pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>n 1</td>
<td>—</td>
<td>e 1</td>
</tr>
<tr>
<td>n 2</td>
<td>—</td>
<td>e 4</td>
</tr>
<tr>
<td>n 3</td>
<td>e 3</td>
<td>e 2</td>
</tr>
<tr>
<td>n 4</td>
<td>e 1</td>
<td>—</td>
</tr>
</tbody>
</table>

### Edge Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Sink</th>
<th>Next Succ</th>
<th>Next Pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>e 1</td>
<td>n 4</td>
<td>n 1</td>
<td>e 2</td>
<td>e 3</td>
</tr>
<tr>
<td>e 2</td>
<td>n 4</td>
<td>n 3</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>e 3</td>
<td>n 3</td>
<td>n 1</td>
<td>e 4</td>
<td>—</td>
</tr>
<tr>
<td>e 4</td>
<td>n 3</td>
<td>n 2</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Implementation Plan

Three Key Steps

• Rename to virtual registers to avoid anti-dependences and constraints based on source register names \textit{(reuse code from lab 2)}

• Build the dependence graph
  – \textit{Specifics of serialization edges may depend on whether or not your scheduler tries to simplify the graph}

• Schedule the operations

One Significant Optimization

• Simplify the graph by eliminating IO serialization edges

• Prove that the memory addresses of IO operations are distinct
  – \textit{Track loadI values, as in rematerialization}
  – \textit{Some students will go so far as to track values computed from rematerializable values}
Implementing the Scheduler

Straightforward to implement, hard to get performance

Implementation Choices

- Is the Ready queue a priority queue or a list or a set of lists?
  - Priority queue is asymptotically faster, don’t know if it gets long enough to see the difference
  - Separate queues for loads & stores, multiplies, & all other operations?

- Is Active one queue or many lists?
  - Fundamental tradeoff in time, space, & ease of programming

- How does your lab break ties?
  - Many, many options

- How does your lab update check for “ready successors”?

---

Cycle ← 1
Ready ← leaves of D
Active ← Ø

while (Ready ∪ Active ≠ Ø)
  for each functional unit, f, do
    if ∃ 1 or more op in Ready for f then
      remove highest prio. op for f from Ready
      S(op) ← Cycle
      Active ← Active ∪ op

Cycle ← Cycle + 1

for each op in Active
  if (S(op) + delay(op) ≤ Cycle) then
    remove op from Active
    add the ready successors of op to Ready
  if ((op is load or store) & S(op) = Cycle -1)
    add the ready store succs of op to Ready
Debugging

The instruction scheduler is complex

• Use graphviz to look at the dependence graph
  – Only useful on small graphs
  – Most of the report blocks are small enough for graphviz
digraph DG {
    18 [label="18: load 1024 => vr2
           prio: <21,21,5>"];
    19 [label="19: load 1024 => vr0
           prio: <16,16,4>"];
    20 [label="20: load 100 => vr4
           prio: <21,21,5>"];
    21 [label="21: load 200 => vr3
           prio: <16,16,4>"];
    23 [label="23: store vr4 => vr2
           prio: <20,20,4>"];
    24 [label="24: store vr3 => vr0
           prio: <15,15,3>"];
    26 [label="26: load vr2 => vr1
           prio: <10,10,2>"];
    27 [label="27: store vr1 => vr0
           prio: <5,5,1>"];
    30 [label="30: output 1024
           prio: <0,0,0>"];
    23 -> 20 [ label="vr4, 1"];
    23 -> 18 [ label="vr2, 1"];
    24 -> 21 [ label="vr3, 1"];
    24 -> 19 [ label="vr0, 1"];
    24 -> 23 [ label="IO Edge, 1"];
    26 -> 18 [ label="vr2, 1"];
    26 -> 24 [ label="IO Edge, 5"];
    27 -> 26 [ label="vr1, 5"];
    27 -> 19 [ label="vr0, 1"];
    27 -> 24 [ label="IO Edge, 1"];
    30 -> 27 [ label="IO Edge, 5"];
}
Generating A Dot File

That seems like a lot of work!

• It is actually quite simple
• 41 lines of C code in lab3_ref

```c
void DGDump() {
    struct Node *nptr;
    struct Edge *eptr;

    fprintf(DOTFILE,"digraph DG {
    DGNodeWalker(ListOfNodes);
    DGEdgeWalker(ListOfEdges);

    fprintf(DOTFILE,"} \n");
    fflush(DOTFILE);
}

static void DGNodeWalker(struct Node *np) { /* DGDump helper fn */
    if (np != DGNullNode) {
        DGNodeWalker(np->NextNode);
        fprintf(DOTFILE, "%d [label="%d: ", np->Op->Line,np->Op->Line);
        PrintOp(np->Op);
        if (PrintPriorities != 0)
            fprintf(DOTFILE,"\n nprio: <%d,%d,%d>",np->Priority,np->Prio2,
                     np->Prio3);
        fprintf(DOTFILE,"\n");
    }
}

static void DGEdgeWalker(struct Edge *ep) { /* DGDump helper fn */
    int latency;
    if (ep != DGNullEdge) {
        DGEdgeWalker(ep->NextEdge);
        if (ep->VReg != DELETED_EDGE) {
            fprintf(DOTFILE," %d -> %d",
                    ep->From->Op->Line,
                    ep->To->Op->Line);
            if (ep->VReg > -1)
                fprintf(DOTFILE," [ label=" sr%d, %d"];\n",
                          ep->VReg,ep->To->Latency);
            else {
                if (ep->From->Op->Opcode == STORE)
                    latency = 1;
                else
                    latency = ep->To->Latency;
                fprintf(DOTFILE," [ label=" IO Edge, %d"];\n",latency);
            }
        }
    }
    fflush(DOTFILE);
}
```
Debugging

The instruction scheduler is complex

• Use graphviz to look at the dependence graph
  – Only useful on small graphs
  – Most of the report blocks are small enough for graphviz

• Write a script that tests it, routinely, against the available blocks
  – The “report” blocks are pretty good; the contributed blocks are also good.

• Checkpoint it regularly
  – GitHub is your friend: Create branches for different heuristics

• Use the trace option on the simulator to understand executions
  – Compare the values that reach specific operations

• Program defensively – write code that checks your internal state
  – Exhaustive graph-checker to verify that the graph was well formed; it tracked down two subtle bugs in the reference implementation.
  – Dump dot files both before & after running the simplifier