Assignment 1 Comments: CCP

1. Lots of people missed the general case
2. Some people also applied this to op_holds: correct, but we hope you considered the semantics carefully before doing this (some people told us they did through comments, others did not so we don't know)
3. Very few handled nops on wire between op_eq and op_eta (only those that I told to) (Tim's fault?)
4. No one (so far) handled more complex boolean conditions (x==10 && y==5)
5. Text descriptions in emails are good but comments in your code are much better.

Assignment 1 Comments: ADCE

1. Many looked at our DCE code, some did not. That is fine, our bad.
2. Return value: true vs. false (many people return true, as our code did)
   - What gets marked live initially?
     - The following are enough, for what you should see in this class: op_ret, op_cal, op_str, op_arg (why?), op_frame, op_pop.
     a) some people used Procedure::sideEffects which is good (shows that you looked through the code), but this function (and most of you) missed two subtleties: are op_args live? are loads live?
     b) some people started by marking all epilog nodes alive (small that's it). could this be an overkill in many cases, where epilog contains stuff other than op_ret?
     c) everyone not using sideEffects missed op_frame and op_pop (that's ok, we did not tell you about them). It does not matter in the trivial adce_example (e.g., what about any nodes?)
   - everyone not using sideEffects missed op_frame and op_pop (that's ok, we did not tell you about them). It does matter In the trivial adce_example (e.g., what about any nodes?)
3. Loads and token wires: when traversing the graph, loads should not be marked live if you reach them through the token wire.

Outline

- c6x review - resource constraints, etc. (continuation of other slides)
- c6x calling convention
- Pegasus massaging for scheduling
- Register alloc on Pegasus
  - forced coalescing
  - spilling / callee-saves
  - phase ordering
- Instruction selection?
- Discussion
- Assignment 2 details
  - adapting Leupers
  - useful tools
Outline

- Pegasus
- C6X
- Scheduling
- Reg Alloc

c6x calling convention

- Our simplifying assumptions:
  - no data segment, and don't need to change FP
  - assume all args fit into available regs
  - each arg fits into a 32b register
  - return value fits into a 32b register
  - assume leaf function?
- Then:
  - args are in A4, B4, A6, B6, ...
  - return PC is in B3
  - return value is in A4
  - stack pointer (SP) is B15
  - SP must always be aligned on 8-byte boundaries!

C6X calling convention

- Prolog
  - decrement SP for locals and spilling
    - addk -88, B15
- Epilog
  - increment SP by same constant
  - branch to B3
    - b B3
    - ...nop...
    - addk 88, B15

Pegasus massaging for Scheduling

- Tokens
- Return PC (Program Counter)
- Stack Pointer
- One Eta per value
- CRT Eta -> branch
- Optimizing branch delay slots!
- Return value
Pegasus massaging for Scheduling

- One Eta per value - when basic blocks

+crt<i+crt

crt eta becomes branch; crt mu becomes label
Pegasus massaging for Scheduling

- crt eta becomes branch; crt mu becomes label
- but must make sure all data is available!

Register allocation on Pegasus

- Register alloc on Pegasus
  - “forced coalescing”
    - webs (mu inputs)
    - special instructions: mux, mvkh
Register allocation on Pegasus

- Register alloc on Pegasus
  - "forced coalescing"
    - webs (mu inputs); the mu itself could be a move, so inputs not coalesced (forcibly) with output

Register allocation: callee saves

- (David's slide)

After Instruction Selection

```assembly
.text
.align 4
.globl _main
_main:
pushl %ebp
movl %esp, %ebp
movl %edi, t2
movl %ebx, t3
movl %esi, t4
movl $2, t7
imull t7, t11
movl t11, t10
imull t10, t8
movl t7, t17
addl $1, t17
movl $33, %eax
cltd
idivl t17
movl %eax, t15
movl t7, t14
addl t15, t14
movl t8, t13
imull t14, t13
movl t14, t13
movl t13, t8
movl $78, t20
negl t20
movl t8, t19
subl t20, t19
movl %eax, %ebp
ret
```
Register allocation on Pegasus

- Register alloc on Pegasus: **callee-saves**
  - not passed explicitly through procedure! regalloc must know.

entry  \( \text{mv A10, t10} \)

exit  \( \text{mv t10, A10} \)

make t10 interfere w/ every def!

**Register allocation: Dumb spilling**

- After scheduling!

Entry block:

Exit block:

Int address: hack meaning stack offset

\[ \text{add B2, B3} \rightarrow B4 \]
\[ \text{stw A12, ++B15[4]} \]

\[ \text{lod ++B15[4], B4} \]
\[ \text{pop} \]
\[ \text{pop} \]
\[ \text{pop} \]
\[ \text{pop} \]
\[ \text{add B4, B5} \rightarrow B6 \]
Phase Ordering

- For spilling (especially callee-saves): maybe better:
  - automatically spill callee saves
  - schedule
  - reg alloc
  - spill
  - compact

  reschedule, knowing about anti- and output deps

Building the interference graph

- Given liveness information, we can build the interference graph (IG)

```plaintext
v ← l
w ← v + 3
x ← w + v
u ← v
u ← v
w ← u + x
x ← w
x ← w
x ← w
u ← u
v ← v
w ← w
x ← x
u ← u
```

How?

Building the interference graph: VLIW?

- Given liveness information, we can build the interference graph (IG)

```plaintext
foreach LIW I working backwards
  foreach op in I
    make \( \text{def}_o \) interfere with liveness
    liveness = liveness \( \setminus \text{def}_o \)
    liveness = liveness \( \cup \text{use}_o \)
```
Building the interference graph: VLIW?

- Given liveness information, we can build the interference graph (IG)

```plaintext
foreach LIW I working backwards
    foreach op in I
        make def<sub>op</sub> interfere with liveness
    foreach op in I
        liveness = liveness \ def<sub>op</sub>
    foreach op in I
        liveness = liveness U use<sub>op</sub>
```

Compaction?

- Mobility is limited due to hard reg assignments
- But can clean up sloppy stuff

Assignment 2 Details

- Adapting Leupers
- Useful tools
- Versions of list scheduling

Assignment 2 Details

- Adapting Leupers - partitioning
  - We can't change args - fixed
    - Could keep on keep same side throughout procedure, or could allow shifting
  - Local vars: a var's web and involved nodes must be assigned to a side (var, etas, mus, constrains) as a unit; each is a "move" candidate
  - Then, remaining operations are assigned individually
  - Mahim's experience!
Assignment 2 Details

- "Free" greedy scheduler that we provide:
  - partition
  - assign sides to args & local variables
  - fix operations to a side due to variables
    (only at circuit/hyperblock/bblock boundaries)
  - then, schedule each circuit via list scheduling -
    some ops are constrained to a side, some are not.

Assignment 2 Details

- Useful tools
  - ASCII dump: Scheduler::print() (in schedasm.cc)
  - .dot dump:
    newdot(c, "reg", &dotsched);
    levelized by cycle, color-coded by function unit
    (gray things are not allocated or are coalesced moves)
    unfortunately, not lined up vertically by function unit...
  - Test cases and submission details to follow
    don't be surprised if a CVS update will follow...sorry!

Assignment 2 Details

- List scheduling
- We do this:
  1. get next best op on ready list
  2. find earliest possible cycle i
  3. try each FU in cycle i to see if possible
  4. if no match, increment i and go to 3
  5. goto 1