Agenda

- The Mystery OOO Processor
- Lab 3
  - The BOOM Processor
  - Lab Questions
  - Demo
DEC Alpha 21264

- 1996/1997
- single-core
  - 4-way
  - out-of-order
  - highly speculative
  - 7-stage
  - up to 80 instructions in flight
  - tournament branch predictor
- 15.2M transistors
  - 6M for logic
  - rest is caching, history tables
- 350 nm
- 600 MHz
- 64KB I$, 64KB D$ (on-chip)
  - 1 to 16MB L2$ (off-chip)
- 314mm² die (fairly large)
DEC Alpha 21264

- 1996/1997
- single-core
  - 4-way
  - out-of-order
  - highly speculative
  - 7-stage
  - up to 80 instructions in flight
  - tournament branch predictor
- 15.2M transistors
  - 6M for logic
  - rest is caching, history tables
- 350 nm
- 600 MHz
- 64KB I$, 64KB D$ (on-chip)
  - 1 to 16MB L2$ (off-chip)
- 314mm$^2$ die (fairly large)
DEC Alpha 21264
Example Machine: MIPS R10k

- 0.35 micron process
- 16.6 x 17.9 mm chip
- 298 mm²
- 6.8 million transistors
  - 4.4 million cache
  - 2.4 million logic
- Full-custom design for datapaths and control logic
- Semi-custom design for less critical control logic

Jan 1996 (150-200 MHz) $3,000 for 200 MHz
1997 -> 250nm @ 250 MHz

R10k Pipeline Timing Diagram

Figure 2. R10000 block diagram (a) and pipeline timing diagram (b). The block diagram shows pipeline stages left to right to correspond to pipeline timing.
The Berkeley Out of Order Machine (BOOM)

- implements RISC-V ISA (RV64S)
- single issue
- full branch speculation (BHT)
- non-blocking data cache
- no HW floating point
- supports supervisor mode, exceptions (boots proxy kernel)
- no virtual memory (can be added if needed)
- ~4,000 loc in Chisel

**default parameters:**
- 16 ROB entries
- 4 issue slots
- 4 LD/ST entries
- 4 entries in fetch buffer
- up to 4 branches
• Directed Portion
  – Gather results from BOOM
  – instrument test-harness and make design recommendations

• Open-ended Portion
  – Your pick of either Chisel, C++, or C

• I’m still writing it/proofing... I apologize for any difficulties that arise from this
  – Directed Portion is ready
  – Open-ended
    • more helpful writeups
    • more diagnostics provided
    • additional benchmark(s)
The BOOM Processor

- Core (BOOM)
- I$
- D$
- Arbiter
- Tile
- Coherence Hub
- Emulated DRAM
- Chisel
- C++ Test Harness (emulator.cpp)

Friday, March 8, 13
The Deep Dive

- implements RISC-V ISA (RV64S)
- single issue
- full branch speculation (BHT)
- non-blocking data cache
- no HW floating point
- supports supervisor mode, exceptions (boots proxy kernel)
- no virtual memory (can be added if needed)
- ~4,000 loc

default parameters:
- 16 ROB entries
- 4 issue slots
- 4 LD/ST entries
- 4 entries in fetch buffer
- up to 4 branches
• Fetch one word at a time
• Puts fetched instructions into *Fetch Buffer*
• uses BHT to predict next instruction
• BTB disabled for this lab
**Decode/Rename**

- **Decode** generates micro-ops (uops)
  - stores broken into StAddrGen, StDataGen
- **Decode** allocates resources
  - ROB entries, BR entries, LSU entries
- **Rename** gives new physical register to every destination register
- **Rename** reads out current mappings for source registers
- In **Rename**, branches take snapshot of the maptables & freelists
Branching

- Every uop has a branch_mask
  - if bit#3 is set, then “I depend on branch #3”
- when a branch is resolved...
  - branch outcome is broadcast to entire machine
    - if misspredict....
      - if uop has bit set for that branch, then uop is killed
      - rename tables are reset to the snapshot associated with that branch
    - if correct prediction
      - uop clears the bit for that branch
      - Decode can now allocate a new branch as “branch #3”
  - branches misspredicts are fixed in a single cycle
Issue & Register Read

- **Uops** sit in the “Issue Window”
  - request to issue if both source operands are ready
- **Issue** selects a ready **uop**, sends to **Register Read**
- In **Register Read**, **uop** reads out its physical source operands
BOOM: A Single Issue Slot

(From the register file's two write ports)

UOP Code \hspace{1cm} BrMask \hspace{1cm} Ctrl... \hspace{1cm} Val \hspace{1cm} RDst \hspace{1cm} RS1 \hspace{1cm} p1 \hspace{1cm} RS2 \hspace{1cm} p2

<table>
<thead>
<tr>
<th>WDest0</th>
<th>=</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

issue slot is valid

resolve or kill

\begin{align*}
\text{Br Logic} & \\
\text{Resolve or Kill} & \\
\end{align*}

<table>
<thead>
<tr>
<th>WDest0</th>
<th>=</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ready

\begin{align*}
\text{RS1} & = \\
\text{p1} & = \\
\text{RS2} & = \\
\end{align*}

\begin{align*}
\text{UOP Code} \rightarrow \text{BrMask} \rightarrow \text{Ctrl...} \rightarrow \text{Val} \rightarrow \text{RDst} \rightarrow \text{RS1} \rightarrow \text{p1} \rightarrow \text{RS2} \rightarrow \text{p2} \rightarrow \text{Br Logic} \rightarrow \text{Resolve or Kill} \rightarrow (\text{From the register file's two write ports}) \rightarrow \text{request} \rightarrow \text{issue}
\end{align*}

Issue Select Logic

Issued to the Register Read stage

Control Signals

Physical Destination Register

Physical Source Registers

Friday, March 8, 13
each instruction gets a br mask... allows us to kill instructions

the register file has two write-ports, so watch both ports’ write addresses
(actually, the ALU WDest can be issued immediately to allow back-to-back dependencies to issue)

each slot asserts request when ready to fire

one slot gets the “issue”

(note: I show a bus implementation, but it’s actually implemented with a bunch of muxes)

Friday, March 8, 13
Execute, Memory, & Writeback

- **Execute**
  - uop is steered to the correct functional unit
  - ALU can issue back-to-back instructions (if ALU bypassing is enabled)
    - “wakeup” is broadcast upon issuing ALU op
  - Memory and Mul/Div/Rem unit share a “variable latency writeport”
    - “wakeup” is broadcast upon writeback to registerfile
  - Mul/Div/Rem unit is unpipelined
• Branches!
  – **any uop anywhere** can be killed at **anytime**
  – even correct predictions require **every uop** to listen
**LSU Behavior**

1. **Incoming loads are immediately issued** to datacache
   - must search SAQ for matches
   - kill mem request on SAQ match
   - forward store data if available

2. If load is killed or nacked (but no store data is forwarded) load sleeps until commit time
   - reissue "sleeper" load at commit

3. Issue stores to memory at commit
   - search LAQ for matches (ordering failures)
   - failed loads require pipeline rollback
**LSU Behavior**

1. **Incoming loads are immediately issued** to datacache
   - must search SAQ for matches
   - kill mem request on SAQ match
   - forward store data if available

2. If load is killed or nacked (but no store data is forwarded) load sleeps until commit time
   - reissue "sleeper" load at commit

3. Issue stores to memory at commit
   - search LAQ for matches (ordering failures)
   - failed loads require pipeline rollback
**Incoming Load Address!**

**LSU Behavior**

1. **Incoming loads are immediately issued to datacache**
   - must search SAQ for matches
   - kill mem request on SAQ match
   - forward store data if available

2. If load is killed or nacked (but no store data is forwarded) load sleeps until commit time
   - reissue "sleeper" load at commit

3. Issue stores to memory at commit
   - search LAQ for matches (ordering failures)
   - failed loads require pipeline rollback

---

**Forwarding Store Data to a Load?**

- Youngest store idx load depends on
- Age load matches
- add matches
- Store Address
  - load executed (sent to mem)
  - load requesting wakeup
  - load failed (detected mem ordering failure)
  - bit mask of dependee stores
  - index of store the load got its data from (if any)

**Data cache**

- Fast/optimistic load
- Sleeper/retry load
- req kill
- Data
- KILL
- fwd_std_val
- fwd_std_idx
- reg kill

**SDQ**

- Data
- val
- data
- KILL
- br kill
- fwd_std_val
- fwd_std_idx

**SAQ**

- Age
- val
- addr
- assoc search

**LAQ**

- addr
- val
- A
- E
- R
- F
- st_mask
- fwd_std_idx

**Register File**

- Writeport

---

Friday, March 8, 13
LSU Behavior

1. **Incoming loads are immediately issued** to datacache
   - must search SAQ for matches
   - kill mem request on SAQ match
   - forward store data if available

2. If load is killed or nacked (but no store data is forwarded) load sleeps until commit time
   - reissue "sleeper" load at commit

3. issue **stores** to memory at commit
   - search LAQ for matches (ordering failures)
   - failed loads require pipeline rollback
Any Questions?
Some of BOOM Parameters

- # of entries in ROB
- # of entries in Issue Window
- # of entries in LD/ST queues
- # of entries in BHT
- # of physical registers
- can turn on/off branch prediction (also change out predictors)
- enable ALU bypassing
- enable speculative (fast) loads
- enable data prefetching

- unfortunately, can’t guarantee that all combinations will pass all tests!
Running BOOM

• Add tools to your path
  $ source ~cs152/sp13/cs152.bashrc

• Copy Lab Files
  $ cp -R ~cs152/sp13/Lab3a ./Lab3

• Build a BOOM Processor, Compile Simulator, Run all Tests & Benchmarks
  $ cd Lab3/
  $./runall.sh

• Takes ~10 minutes
Lab 3: Directed Portion

• Question 1
  – collect CPI with BHT, without, compare to 5-stage in-order

• Question 2
  – start at worst case, improve

• Question 3
  – Probe the Instruction Window to potential benefit of dual issue

• Question 4
  – Probe IW for dual issue of ALU/Mem ops
• Question 1
  – collect CPI and BrPred accuracy, without and without a BHT
    • compare to 5-stage in-order (results provided)
Lab 3: Directed Portion

• Question 2
  – collect **CPI**
  – start with *no* features, slowly add back...
    • 33 -> 64 registers
    • add back branch predictor
    • allow load speculation (otherwise execute at **commit**)
    • enable ALU bypassing

  – **note**: if **dhrystone** benchmark fails, that’s just a **timeout** error. That’s okay, it’s just slow.
Lab 3: Directed Portion

• Question 3
  – instrument the issue window
  – write C++ in the test harness to measure how often >1 uops request to be issued
  – how much performance are we losing by not going dual-issue?

• Question 4
  – what if we issued ALU+Mem?
BOOM: A Single Issue Slot

(From the register file's two write ports)

(UOP Code) > BrMask > Ctrl... > Val > RDst > RS1 > p1 > RS2 > p2

issue slot is valid

Br Logic
Resolve or Kill

WDest0
WDest1

issue

request

Control Signals
Physical Destination Register
Physical Source Registers

Issued to the Register Read stage
each instruction gets a br mask... allows us to kill instructions

the register file has two write-ports, so watch both ports’ write addresses*
(Actually, the ALU WDest can be issued immediately to allow back-to-back dependencies to issue)

each slot asserts request when ready to fire

one slot gets the “issue”

uop holds the micro-op code (is it a LD, an ADD, etc.)

Control Signals

Physical Destination Register

Physical Source Registers

Issued to the Register Read stage
Probing Chisel from C++

- These signals are declared in emulator/riscv-boom/generated-src/Top.h
- `dat_t<1> Top_SodorTile_core_d_IntegerIssueSlot__slot_valid`
- `dat_t<1> Top_SodorTile_core_d_IntegerIssueSlot_1__slot_valid`
- `dat_t<1> Top_SodorTile_core_d_IntegerIssueSlot_2__slot_valid`
- `dat_t<1> Top_SodorTile_core_d_IntegerIssueSlot_3__slot_valid`

- These are Chisel `dat_t` types. To get to `int64_t`...
- `Top_SodorTile_core_d_IntegerIssueSlot__slot_valid.lo_word()`

- Grep emulator/riscv-boom/generated-src/Top.h for Chisel signals
Open-ended

• A few of them are still in flux
  – I may add another Question to pick from
  – 3.5 will have stuff added to it (writing benchmarks)

• With that said...
Open-ended: Options

• Option 1
  – Build a better branch predictor for BOOM (Chisel)
  – compete against your classmates (and me...)

• Option 2
  – Build a data prefetcher
  – improve CPI, lower cache miss rate
  – compete against your classmates (but not me...)

• Option 3
  – Part A: write micro-benchmarks to torture the LSU
  – Part B: pick a machine parameter and try to introspect it

• Option 4
  – write a branch predictor in C++

• Option 5
  – instrument the LSU, to analyze a design decision I made and recommend changes? (I may remove this question)
• How do prefetchers work?
  – on miss... fetch N-nearest blocks
  – recognize strided accesses
  – recognize strided-segment accesses
  – combine a collection of prefetchers and predict which one(s) is most likely correct

• Watch out for over-fetching!
• Option 1
  – improve CPI, lower cache miss rate

• Note
  – I will provide an additional benchmark for this question (lots of strided accesses)
  – a bug in the incoming address stream, I hope to update soon
Branch Predictors
Two kinds of correlating branch predictors:

- **Local**
  - Local History Table
  - Branch History Table
  - PC

- **Global**
  - Branch History Table
  - Global History
• 21264 uses both! (tournament predictor)
Tournament Branch Predictor
(Alpha 21264)

- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications
Debugging And Monitoring

• runall.sh **debug**
  – prints out +verbose **text output**
    • saved to .tar.gz to save space
    • zcat vvadd.tar.gz | vim -
      – to open in vim
  – prints out **.vcd waveform**
    • dve is provided on inst machines
  – add your own counters to OOOTracer.cpp
    • count events, etc.
Reading Printouts
Reading Printouts

FetchBuffer

ROB

LAQ

SAQ

SDQ

PCs

Issue Window

Friday, March 8, 13
Reading Printouts

- FetchBuffer
- ROB
- LAQ
- SAQ
- SDQ
- freelist (bit-array of free/notfree bools)
- new physical register dest given out
- preregister valid?
- BHT Prediction?
- Issued Inst?
- Inst in Decode
- ISA registers
- Translated Physical Registers
- Branch Resolution

PCs

Friday, March 8, 13
comment out “#define DEBUG” in emulator.cpp to get the “short form” debugging; turn off -DDEBUG in common/Makefile.include

- PCs (all 6-stages)
- Micro-ops (all 6-stages)
- Issued Inst?
- branch resolution
- IW full?
Analyzing the Waveforms

- Although not necessary to complete the lab, it is possible to generate waveform files from Chisel runs (stored in cpu.vcd file).
• Your code is a Scala program
  – Chisel is a library
  – “Running” your Chisel code generates C++ (or Verilog)

• Bundles
  – collection of wires

• Components
  – pieces of hardware
  – every component has an IO bundle
Caveats

• BOOM is *brand new*
  – and thus buggy! (maybe)
  – email me if you have tests/benchmarks fail

• Brand new questions!
  – You are running on (pretty) real HW
  – I can’t guarantee performance results will make sense

• I **will** be pushing updates to the statistics code (but won’t break your Directed answers).
Final Notes on Lab 3

• Use Piazza
  – there’s a lot here, it can be intimidating, but don’t let that stop you!

• Work together!

• Have fun!
Cool down Question: Why build OoO?

- excellent at finding ILP in code
- tolerate variable (but short) latencies
- A complex method for precise, fine-grain data prefetching
  - pull out loads and execute ASAP

- **Industry** trend towards more OoO, even in energy-efficient mobile processing
  - ARM CortexA9, CortexA15
  - Intel Atom moving to OoO at 22nm
  - Oracle S3
  - Intel Itanium (Paulson)

- **Aside**
  - most ILP is probably just DLP being lazily written as scalar code
  - a good “ILP processor” is *not* necessarily a good “scalar processor”
If you like Chisel/BOOM...

• This is just the beginning of BOOM
  – I’m looking for excited undergrads who want to hack processors
  – laundry list of features to add
    • wider issue widths
    • higher performance LD/ST unit
    • floating point support
    • AMOs
    • prefetchers (I$ and D$)
    • better branch predictors
    • perform energy/power studies
    • higher performance fetch unit
    • and more...
• implements RISC-V ISA (RV64S)
• single issue
• full branch speculation (BHT)
• non-blocking data cache
• no HW floating point
• supports supervisor mode, exceptions (boots proxy kernel)
• no virtual memory (can be added if needed)
• ~4,000 loc

default parameters:
  - 16 ROB entries
  - 4 issue slots
  - 4 LD/ST entries
  - 4 entries in fetch buffer
  - up to 4 branches