ENGN1640: Design of Computing Systems
Topic 06: Advanced Processor Design

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2016

[material from Harris & Patterson & Hennessy]
Advanced processor design

1. Floating point arithmetic
2. SIMD (data-level parallelism)
3. Superscalar (instruction-level parallelism)
4. Multi-threading HW (thread-level parallelism)
1. Floating points

\[ x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)} \]

S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
Normalize significand: 1.0 ≤ |significand| < 2.0
Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit). Significand is Fraction with the “1.” restored
Example

1. What is the floating point representation of –0.75?

\[-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}\]

S = 1, Fraction = 1000…00\(_2\)
Exponent = –1 + Bias = –1 + 127 = 126 = 01111110\(_2\)
Answer: 101111101000…00

2. What is the decimal value of the following IEEE number

1 01111100 110000000000000000000000

The sign bit s is 1.
The e field contains 01111100= 124.
The mantissa is 0.10100…= 0.625.
\[-1.625 \times 2^{-3}= -0.203125\]
Example of floating point addition

• Consider a 4-digit decimal example
  \[9.999 \times 10^1 + 1.610 \times 10^{-1}\]
• 1. Align decimal points
  Shift number with smaller exponent
  \[9.999 \times 10^1 + 0.016 \times 10^1\]
• 2. Add significands
  \[9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1\]
• 3. Normalize result & check for over/underflow
  \[1.0015 \times 10^2\]
• 4. Round and renormalize if necessary
  \[1.002 \times 10^2\]
Example of floating point addition

• Now consider a 4-digit binary example
  \[1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}\]  
  \[0.5 + -0.4375\]

• 1. Align binary points
  Shift number with smaller exponent
  \[1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}\]

• 2. Add significands
  \[1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}\]

• 3. Normalize result & check for over/underflow
  \[1.000_2 \times 2^{-4}\], with no over/underflow

• 4. Round and renormalize if necessary
  \[1.000_2 \times 2^{-4}\] (no change)  \[= 0.0625\]
Floating point addition HW

Step 1
- Compare exponents

Step 2
- Shift smaller number right
- Add

Step 3
- Normalize

Step 4
- Round

Diagram shows the process of floating point addition, including steps for handling the sign, exponent, and fraction parts.
Comparison of HW area requirements

- 32 LEs for integer 32-bit addition
- 840 LEs for floating point single-precision addition
- Floating point operating have larger delay and take multiple cycles in EX stage.
floating point ISA interface (e.g., ARM)

Floating Point Register File

<table>
<thead>
<tr>
<th>S0</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>S28</td>
<td>D14</td>
</tr>
<tr>
<td>S29</td>
<td></td>
</tr>
<tr>
<td>S30</td>
<td></td>
</tr>
<tr>
<td>S31</td>
<td>D15</td>
</tr>
</tbody>
</table>

Floating Point Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>FMULc/S/D{(cond)}</td>
</tr>
<tr>
<td>negate</td>
<td>FMULc/S/D{(cond)}</td>
</tr>
<tr>
<td>accumulate</td>
<td>FMULc/S/D{(cond)}</td>
</tr>
<tr>
<td>deduct</td>
<td>FMULc/S/D{(cond)}</td>
</tr>
<tr>
<td>negate and accumulate</td>
<td>FMULc/S/D{(cond)}</td>
</tr>
<tr>
<td>negate and deduct</td>
<td>FMULc/S/D{(cond)}</td>
</tr>
<tr>
<td>Add</td>
<td>FADDc/S/D{(cond)}</td>
</tr>
<tr>
<td>Subtract</td>
<td>FSUBc/S/D{(cond)}</td>
</tr>
<tr>
<td>Divide</td>
<td>FDIVc/S/D{(cond)}</td>
</tr>
<tr>
<td>Copy</td>
<td>FCVTc/S/D{(cond)}</td>
</tr>
<tr>
<td>Absolute</td>
<td>FABSc/S/D{(cond)}</td>
</tr>
<tr>
<td>Negative</td>
<td>FNNEGc/S/D{(cond)}</td>
</tr>
<tr>
<td>Square root</td>
<td>FPQRTc/S/D{(cond)}</td>
</tr>
</tbody>
</table>

Scalar compare
- Compare with zero
- Single to double
- Double to single
- Unsigned integer to float
- Signed integer to float
- Float to unsigned integer
- Float to signed integer

Scalar convert
- 

Save VFP registers
- Multiple, unindexed
- Increment after decrement before
- 

Load VFP registers
- Multiple, unindexed
- Increment after decrement before
- 

Transfer registers
- ARM to single
- Single to ARM
- ARM to lower half of double
- Lower half of double to ARM
- ARM to upper half of double
- Upper half of double to ARM
- ARM to VFP system register
- VFP system register to ARM
- FPSR flags to CPSR
- 

Key to Tables
<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{cond}</td>
<td>See Table Condition Field (on ARM side).</td>
</tr>
<tr>
<td>&lt;S/D&gt;</td>
<td>S (single precision) or D (double precision).</td>
</tr>
<tr>
<td>&lt;S/D/X&gt;</td>
<td>As above, or X (unspecified precision).</td>
</tr>
<tr>
<td>Pd, Fn, Fm</td>
<td>Sd, Sn, Sm (single precision), or Dd, Dn, Dm (double precision).</td>
</tr>
</tbody>
</table>
Improving throughput with SIMD and Superscalar

\[
\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}
\]

optimization goal for multiple-issue processors

• Pipeline CPI is less than 1:
  – Data Hazard Stalls
  – Control Stalls
  – Memory cache misses

• Goal of superscalar is to maximize throughput (i.e., IPC \times frequency)
2. SIMD architectures

- Single Instruction Multiple Data (SIMD)
  - Single instruction acts on multiple pieces of data at once (aka vector instructions)
  - Common application: scientific computing, graphics
- Requires vector register file and multiple execution units
- Examples: ARM NEON and Intel SSE2 extensions.

![Diagram showing scalar and vector operations]

**SCALAR**
(1 operation)

- `r1`, `r2` -> `r3`
- `add r3, r1, r2`

**VECTOR**
(N operations)

- `v1`, `v2` -> `v3`
- `add.vv v3, v1, v2`

*vector length*
EXAMPLE: ARM NEON SIMD extensions

- New additional register file \( \rightarrow \) can be viewed as:
  - Sixteen 128-bit quadword registers: Q0-Q15
  - Thirty-two 64-bit doubleword registers: D0-D31

- Data packing inside a register: 16 integer bytes (16x8 bits), 8 integer words (8x16), 8 half-precision FP numbers (8x16), 4 integer doublewords (4x32), 4 single-precision FP numbers (4x32), 2 integer quad words (2x64).

- Examples:
  - \texttt{VADD.I16 Q0, Q1, Q2} ; 16-bit integer elements stored in 128-bit Q registers
  - \texttt{VMULL.S16 Q0, D2, D3} ; 4 16-bit lanes \( \rightarrow \) 4 32-bit products in a 128-bit vector
  - Structured load and store instructions that can load or store single or multiple values from or to single or multiple lanes in a vector register.
3. Superscalar architectures

- **VLIW:**
  - Compiler groups instructions to be issued together Very Large Instruction Words (VLIW)
  - Packages them statically into “issue slots”
  - Compiler detects and avoids hazards

- **Superscalar:**
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime
  - Can be *static in order* or dynamic out-of-order (won’t discuss)
Difference between superscalar and VLIW

[from Fisher et al.]
Multiple copies of datapath execute multiple instructions at once.

Dependencies make it tricky to issue multiple instructions at once.
Superscalar example

Ideal IPC: 2
Actual IPC: 2

Notice that there are no dependencies.
All resolved by compiler → VLIW
Static superscalar with hazard detection

Ideal IPC: 2
Actual IPC: 6/5 = 1.2
Scheduling example

Assume dual-issue ARM

```assembly
MOV R0, #0
Loop:  LDR  R1, [R10, 0]  # R0=array element
       ADD  R1, R1, R12  # add scalar in $s2
       STR  R1, [R10, 0]  # store result
       ADD  R10, R10, -4  # decrement pointer
       CMP  R10, R0
       BNE  Loop
```

- IPC = 6/6 = 1 (peak dual-issue IPC = 2 and single-issue IPC = 6/7 = 0.85)
- Can the compiler do a better scheduling to improve ILP?
Limits to ILP: data dependencies

• Data dependencies determine:
  – Order which results should be computed
  – Possibility of hazards
  – Degree of freedom in scheduling instructions
  => limit to ILP

• Data/Name dependency hazards:
  – Read After Write (RAW)
  – Write After Read (WAR)
  – Write After Write (WAW)
Register hazards from dynamic scheduling

Read After Write (RAW) hazard (true dependency)

<table>
<thead>
<tr>
<th>ADD</th>
<th>R2, R1, R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>R4, R2, R3</td>
</tr>
</tbody>
</table>

- A true data dependency because values are transmitted between the instructions

Write After Read (WAR) hazard (anti dependency)

<table>
<thead>
<tr>
<th>ADD</th>
<th>R4, R2, R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>R2, R1, R3</td>
</tr>
</tbody>
</table>

- Introduced by OOO
- Just a name dependency – no values being transmitted
- Dependency can be removed by renaming registers (either by compiler or HW)

Write After Write (WAW) hazard (output dependency)

<table>
<thead>
<tr>
<th>ADD</th>
<th>R2, R1, R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>R2, R2, R3</td>
</tr>
</tbody>
</table>

- Introduced by OOO
- Just a name dependency – no values being transmitted
- Dependency can be removed by renaming registers (either by compiler or HW)
Re-schedule example

Loop:  LDR  R1, [R10, 0]  # R0= array element
        ADD  R1, R1, R12  # add scalar in $s2
        STR  R1, [R10, 0]  # store result
        ADD  R10, R10, -4  # decrement pointer
        CMP  R10, R0
        BNE  Loop

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>nop</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>LDR  R1, [R10, 0]</td>
<td>2</td>
</tr>
<tr>
<td>ADD</td>
<td>R10, R10, -4</td>
<td>3</td>
</tr>
<tr>
<td>ADD</td>
<td>R1, R1, R12</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>R10, R0</td>
<td>5</td>
</tr>
<tr>
<td>BNE</td>
<td>Loop</td>
<td></td>
</tr>
</tbody>
</table>

- IPC = 6/5 = 1.2 (c.f. peak IPC = 2)
Exposing ILP using loop unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead

- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name
Loop unrolling and renaming

Loop unrolling:

```
Loop: LDR  R1, [ R10, 0]
      ADD  R1, R1, R12
      STR  R1, [ R10, 0]
      LDR  R1, [ R10, -4]
      ADD  R1, R1, R12
      STR  R1, [ R10, -4]
      LDR  R1, [ R10, -8]
      ADD  R1, R1, R12
      STR  R1, [ R10, -8]
      LDR  R1, [ R10, -12]
      ADD  R1, R1, R12
      STR  R1, [ R10, -12]
      ADD  R10, R10, –16
      CMP  R10, R0
      BNE  Loop
```

Loop unrolling + renaming:

```
Loop: LDR  R1, [ R10, 0]
      ADD  R1, R1, R12
      STR  R1, [ R10, 0]
      LDR  R2, [ R10, -4]
      ADD  R2, R2, R12
      STR  R2, [ R10, -4]
      LDR  R3, [ R10, -8]
      ADD  R3, R1, R12
      STR  R3, [ R10, -8]
      LDR  R4, [ R10, -12]
      ADD  R4, R1, R12
      STR  R4, [ R10, -12]
      ADD  R10, R10, –16
      CMP  R10, R0
      BNE  Loop
```
Loop unrolling scheduling

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R10, R10, -16</td>
<td>LDR R1, [ R10, 0]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>LDR R2, [ R10, 12]</td>
<td>2</td>
</tr>
<tr>
<td>ADD R1, R1, R12</td>
<td>LDR R3, [ R10, 8]</td>
<td>3</td>
</tr>
<tr>
<td>ADD R2, R2, R12</td>
<td>LDR R4, [ R10, 4]</td>
<td>4</td>
</tr>
<tr>
<td>ADD R3, R3, R12</td>
<td>STR R1, [ R10, 16]</td>
<td>5</td>
</tr>
<tr>
<td>ADD R4, R4, R12</td>
<td>STR R2, [ R10, 12]</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>STR R3, [ R10, 8]</td>
<td>7</td>
</tr>
<tr>
<td>CMP R10, R0</td>
<td>STR R4, [ R10, 4]</td>
<td>8</td>
</tr>
<tr>
<td>BNE Loop</td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

- IPC = 15/9 = 1.67
  - Closer to 2, but at cost of registers and code size
Limitations of static superscalar and VLIW

- **VLIW:**
  - SW tied to HW.
  - Some dependencies cannot be identified during compilation.

- **Static superscalar:**
  - Branch mispredictions and cache misses will have high impact.
  - RAW dependencies will lead to further stalls.

1. Can we rename and schedule instructions dynamically in HW?
2. Can ready instruction bypass stalled instructions and execute out-of-order (OoO)?
Gist of dynamic superscalar (A57)

Decoder:
- Breaks more complex instructions into uops
- Detects dependencies
- Renames registers to avoid WAR and WAW hazards
- Dispatches instructions (max 3) to appropriate queues

Issue Queue:
- Instructions wait until ready (i.e. all operands are available and forwarded)
- Once ready, instructions are issued out-of-order once EX unit is available

WB:
- Results are forwarded to any instructions waiting for operands in queue
- Results are written in order in the register file
Summary of superscalar architectures

- **Pros**: Improved single-thread throughput
- **Cons**:
  - impacts silicon area
  - impacts power consumption
  - impacts design complexity (especially for dynamic OoO)

SW compilation techniques enable
- more ILP from the same HW
- simplify HW (e.g., VLIW) at the expense of code portability
4. Multi-threaded and multi-core processors

- Moore’s Law + Superscalar limitations → Multi-threaded and Multi-core processors.
- Improves total throughput but not single thread performance.
- Applications need to be re-coded to use the parallelism.

**Topics:**

A. Software multi-threading.
B. HW block multi-threading.
C. HW interleaved multi-threading.
D. Multi-core processors.
A. Software multi-threading

• Used since the 1960’s to enable multiple users to use the same computing system and to hide the latency of I/O operations and

• Creating illusion of parallelism for multiple processes/threads using **context switching**:
  1. Trap processor -- flush pipeline
  2. Save process state (includes register file(s), PC, page table base register, etc) in OS process control block which is in memory.
  3. Restore process state of a different ready process
  4. Start execution -- fill pipeline

• On an I/O operation (or shared resource conflict or timer interrupt)
  1. Process is preempted and removed from ready list
  2. I/O operation is started
  3. Another active process is picked from the ready list and run
  4. When I/O completes → preempted process back in the ready list

• Very high switching overhead (ok, since wait is very long)
Hardware multithreading

- Vertical waste (e.g., cache misses) and horizontal waste (ILP limits) can limit potential of superscalar processors.
- Fill the waste slots from instructions of other threads.
- HW runs multiple threads concurrently → OS thinks there is more than one processor, each processing its own thread.
- HW threads share cache memory → enables rapid communication by threads → fine-grain communication by threads.
B. Block multithreading (coarse grain)

- Core has multiple contexts → A hardware context is the program counter, register file, and other data required to enable a software thread to execute on a core.
- Each running thread executes in turn until a long latency event → switch out by flushing and load context of another thread.
- Similar to software multithreading but with HW support for fast context switch → order less than 10 cycles versus thousands of cycles.
- Block multithreading can hide long cache misses. However, it typically cannot hide short inter-instruction latencies.
Required changes for block multi-threading

1. Fetch:
   - Multiple PCs; selector base on active thread
   - Miss in caches trigger context switching
2. Read:
   - Need to replicate the register file \(\rightarrow\) might impact RF latency
3. Ex: Nothing
4. MEM:
   - data memory needs to be non-blocking
5. WB:
   - Need to know the thread id.
Example of operation

- T0: add
  - Instruction Fetch (IF)
  - Read Data (RD)
  - ALU
  - Memory (MEM)
  - Write Back (WB)

- T0: load (cache miss)
  - IF
  - RD
  - ALU
  - MEM
  - (buffered)

- T0: sub
  - IF
  - RD
  - ALU
  - (squashed)

- T0: add
  - IF
  - RD
  - (squashed)

- T0: or
  - IF
  - (squashed)

- Cxt swap
  - Instructions for context switch

- T1: sub
  - IF
  - RD

- T1: beq
  - IF
C. Interleaved (fine-grain) multi-threading

- Dispatch instructions from different threads/processes in each cycle
  - Able to switch between contexts (threads) as every cycle with no switch delay.
  - Takes advantage of small latencies such as instruction execution latencies
  - Does not address horizontal waste but can eliminate vertical waste
Interleaved architecture example

- Two threads: penalty of a taken branch is one clock!
- Five threads: penalty of a taken branch is zero!
Interleaved multi-threading architecture

Same architecture as for block multithreading except that:

- Data forwarding must be thread aware
- Context id is carried by forwarded values
- Stage flushing must be thread aware
- On an miss exception IF, ID, EX and MEM cannot be flushed indiscriminately
- Same for taken branches and regular software exceptions
- Thread selection algorithm is different: a different thread is selected in each cycle (round-robin)
- On a long latency event the selector puts the thread aside and removes it from selection
Impact of increasing number of contexts

- Fetch can be simplified $\rightarrow$ no need for branch prediction and control hazard circuit
- Forwarding can be eliminated and no need for data hazard hardware
- Issues:
  - Single-thread performance can suffer
  - Register file needs to increase proportionally.
  - Interference in cache.
D. Multi-core processors

- Multiple independent cores
- Hardware provides single physical address space for all processors → shared memory is used for communication
- Big problem is cache coherency
- Interconnection network: enables cores to share items in their caches.
Example: Sun/Oracle UltraSparc T1

- Targets server market for transactional applications → high TLP
- T1 (introduced in 2005) 8 cores @ 1.4 GHz – 72W
- Each core supports four thread contexts → total 32 HW threads → OS sees 32 virtual/logical cores
- Each core is in-order scalar core with simple 6-stage pipeline: fetch, thread select, decode, execute, memory, writeback.
- L2 cache shared among all cores using a cross bar
- Static branch prediction using a hint bit
Summary

1. SIMD exploits data-level parallelism → needs compiler intervention to explicitly use vector instructions.
2. Floating point and SIMD require register file and ISA extensions
3. Superscalar exploits instruction-level parallelism → no need for recompilation though compilation help improve performance and mitigate pressure on HW
4. Multi-cores exploits thread-level parallelism → needs programmer intervention to re-write the code.