ELEC6234
Embedded Processor Design

Hardware multithreading
Multiple threads

- Modern Operating systems commonly support threads (i.e. lightweight processes) and many modern processors are multi-core shared memory architectures
- Multithreading is one method for sharing the work of a single application across multiple processors with shared memory
- Multithreading also helps to prevent deadlocks due to unpredictable behaviour of input events or unsafe coding.
- Software vs hardware multithreading
  - Software: multithreading without dedicated hardware support for storing state (PC, registers, etc.) for multiple threads simultaneously; stacks in memory are used for that purpose and handled by software
  - Hardware: hardware-level support for storing state for multiple threads, permitting fast context switches
  - A major argument for multithreading: memory bandwidth utilisation (see next slide)
Memory latency, bandwidth and multithreading

• Typical memory latencies:
  – L1: 2-4 cycles; L2: ~10 cycles; DRAM: ~200 cycles

• Latencies in shared-memory multiprocessors
  – Remote memory accesses cost much more than local ones
  – Remote transactions and cache coherence cost is high: 10-100x reduction of performance

• Memory bandwidth: the rate at which data can be read from or stored into a memory by a processor.
  – Expressed in units of bytes/second

• From queuing theory: the average maximum number of memory words “in flight” between memory and processor is the product of latency and bandwidth:
  – concurrency = bandwith * latency

• Major argument for multithreading and multiple cores accessing a single memory
  – Multiple threads can hide the memory latency
Coarse grained multithreading

• Usually used in pipelined architectures
• Single thread runs until a costly stall
  – e.g. a I-cache or a D-cache miss
• Another thread starts during stall for first
  – Note that pipeline fill time requires several cycles
• Hardware support:
  – PC and registers need to be saved for each thread
  – Small amount of control hardware
### Switching threads on I-cache miss

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst b</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst c</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst X</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst Y</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst Z</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- At cycle 3 remove Inst c and switch to other thread
- The next thread will continue its execution until there is another I-cache or D-cache miss
## Switching threads on D-cache miss

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M-Miss</td>
<td>MISS</td>
<td>MISS</td>
<td>MISS</td>
</tr>
<tr>
<td>Inst b</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inst c</td>
<td>IF</td>
<td>ID</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inst d</td>
<td>IF</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inst X</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inst Y</td>
<td>IF</td>
<td>ID</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- At cycle 4 remove Inst a and switch to other thread
  - Remove the rest of instructions from ‘blue’ thread in the pipeline
  - Roll back ‘blue’ PC to point to Inst a
Coarse Grain Multithreading

- Good to reduce effects of infrequent, but expensive pipeline disruptions
- Minimal hardware overheads
  - D-cache miss causes abort of all the instructions further in the pipeline
  - Resume instruction stream to recover
- Short stalls (data/control hazards) are not solved
Fine grained multithreading

• Two or more threads interleave instructions
  – Round-robin fashion
  – Skip stalled threads in pipelined processors
• Hardware support
  – Separate PC and register file for each thread
  – Hardware scheduler to control alternating pattern
• Naturally hides small delays
  – Data hazards, Cache misses
  – Pipeline runs with rare stalls
Fine-Grain Multithreading

- Multithreading helps to eliminate effects causing short pipeline stalls
- In the example below there is no need for forwarding or interlocking

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst M</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst b</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst N</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst c</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst P</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Inst a: Instructions are processed through the pipeline stages from left to right (IF, ID, EX, MEM, WB).
Inst M: Similar to Inst a, no need for forwarding or interlocking.
Inst b: Again, similar to Inst a and Inst M.
Inst N: Instructions are processed with a slight delay, no forwarding or interlocking needed.
Inst c: Similar to Inst N.
Inst P: No forwarding or interlocking, as all instructions complete in one cycle.
### I-cache misses in Fine Grain Multithreading

- An I-cache miss is overcome easily

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst M</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst b</td>
<td>IF-MISS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst N</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst P</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst Q</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Inst b is removed and the thread is marked as not ‘ready’

- ‘Blue’ thread is not ready so ‘orange’ is executed
D-cache misses in Fine Grain Multithreading

- Mark the thread as not ‘ready’ and issue only from the other thread

```
<table>
<thead>
<tr>
<th>Inst</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M-MISS</td>
<td>Miss</td>
<td>Miss</td>
<td>WB</td>
</tr>
<tr>
<td>Inst M</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst b</td>
<td>IF</td>
<td>ID</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inst N</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst P</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst Q</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Thread marked as not ‘ready’. Remove Inst b. Update PC.

‘Blue’ thread is not ready so ‘orange’ is executed
Fine Grain Multithreading

- Improves the utilisation of pipeline resources
- Many short stalls can be eliminated by executing instructions from other threads
- Data forwarding not needed to eliminate stalls
- Requires hardware support for instantaneous thread switching
Simultaneous Multithreading (SMT)

• Some architectures go beyond pipelining to execute more than one instruction in one clock cycle.
  – Duplicate some of the functional parts of the processor (e.g., have two ALUs or a register file with 4 read ports and 2 write ports), and have logic to issue several instructions concurrently.
  – Two general approaches to multiple issue: static multiple issue (scheduling is done at compile time) and dynamic multiple issue (scheduling is done at execution time), also called superscalar.
  – Intel Core 2 processors are superscalar and can issue up to 4 instructions per clock cycle.

• SMT: instructions from multiple threads are issued at same cycle
  – Uses dynamic scheduling hardware of multi-issue architecture

• Needs more hardware support
  – Register files, PC’s for each thread
  – Temporary result registers before commit
  – Scheduler hardware to link supply correct results to the right threads
  – SMT maximizes utilization of execution units
**Simultaneous Multithreading**

- Multiple instruction issue and SMT

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst b</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst M</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst N</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst c</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst P</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst Q</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst d</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst e</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst R</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instr c & P issued from 2 threads
Simultaneous Multi Threading

• Extracts the most parallelism from instructions and threads
• Implemented only in superscalar out-of-order processors because only such processors are able to exploit that much parallelism
• Has a significant hardware overhead
SMT Performance

Example showing multiple instructions in the pipeline and stalls

Blue Thread

Red Thread

Green Thread

Yellow Thread

3 instructions in pipeline in a single cycle

stall cycles
Coarse multithreading hides some stalls

Long stalls in Blue and Green become shorter
If there are long stalls at end of each thread they become shorter
Fine multithreading interleaves threads and eliminates short stalls

Blue skipped here
Simultaneous multithreading fully utilises resources due to multi-issue hardware
Example of simple multithreaded hardware

- Simple hardware scheduler selects a thread for the pipeline
- Appears to software (including OS) as multiple, albeit slower, CPUs

Source: Onur Mutlu lecture, Carnegie Mellon Uni
Chip Multithreading (CMT)

Thread scheduling on a four-core processor

Hides latency of shared memory

Hardware multithreading summary

• A cost-effective way of exploiting parallelism of a CPU pipeline
• Most architectures present additional CPU thread as additional CPU to Operating System
• Main motivation for this lecture: a pipelined multithreaded RISC-V is being developed in Cambridge for the POETS project (Partially Ordered Event Triggered Systems).
  – The purpose of the POETS research project is to explore massively parallel architectures that might be automatically configured to self-organise according to application.
  – Ron Minnich, Google software developer at Third RISC-V Workshop in Mountain View, Calif. 2016: „RISC-V is the Linux of processor architectures“