Lecture 2:

Pipelining and Instruction-Level Parallelism

15-418 Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2019
Many kinds of processors

Why so many? What differentiates these processors?
Why so many kinds of processors?

Each processor is designed for different kinds of programs

- CPUs
  - “Sequential” code – i.e., single / few threads

- GPUs
  - Programs with lots of independent work → “Embarrassingly parallel”

Parallelism pervades architecture

- Speeding up programs is all about parallelism
  - 1) Find independent work
  - 2) Execute it in parallel
  - 3) Profit

- Key questions:
  - Where is the parallelism?
  - Whose job is it to find parallelism?
Where is the parallelism?

Different processors take radically different approaches

- CPUs: Instruction-level parallelism
  - Implicit
  - Fine-grain

- GPUs: Thread- & data-level parallelism
  - Explicit
  - Coarse-grain
Whose job to find parallelism?

Different processors take radically different approaches

- CPUs: Hardware dynamically schedules instructions
  - Expensive, complex hardware $\Rightarrow$ Few cores (tens)
  - (Relatively) Easy to write fast software

- GPUs: Software makes parallelism explicit
  - Simple, cheap hardware $\Rightarrow$ Many cores (thousands)
  - (Often) Hard to write fast software
Visualizing these differences

- Pentium 4
  “Northwood” (2002)
Visualizing these differences

- Pentium 4
  “Northwood” (2002)

- Highlighted areas actually execute instructions

  ➔ Most area spent on scheduling (not on executing the program)
Visualizing these differences

- AMD Fiji (2015)
Visualizing these differences

- AMD Fiji (2015)

- Highlighted areas actually execute instructions
  - Most area spent executing the program
    - (Rest is mostly I/O & memory, not scheduling)
Today you will learn...

How CPUs exploit ILP to speed up straight-line code

- Key ideas:
  - **Pipelining & Superscalar**: Work on multiple instructions at once
  - **Out-of-order execution**: Dynamically schedule instructions whenever they are “ready”
  - **Speculation**: Guess what the program will do next to discover more independent work, “rolling back” incorrect guesses

- CPUs must do all of this while preserving the illusion that instructions execute in-order, one-at-a-time
In other words... Today is about:
Buckle up!

...But please ask questions!
Example:
Polynomial evaluation

```c
int poly(int *coef,
         int terms, int x) {
    int power = 1;
    int value = 0;
    for (int j = 0; j < terms; j++) {
        value += coef[j] * power;
        power *= x;
    }
    return value;
}
```
Example: Polynomial evaluation

- Compiling on ARM

```c
int poly(int *coef, int terms, int x) {
    int power = 1;
    int value = 0;
    for (int j = 0; j < terms; j++) {
        value += coef[j] * power;
        power *= x;
    }
    return value;
}
```

```
 poly:
            cmp        r1, #0
            ble        .L4
            push       {r4, r5}
            mov        r3, r0
            add        r1, r0, r1, lsl #2
            movs       r4, #1
            movs       r0, #0
            .L3:
            ldr        r5, [r3], #4
            cmp        r1, r3
            mla        r0, r4, r5, r0
            mul        r4, r2, r4
            bne        .L3
            pop        {r4, r5}
            bx          lr
            .L4:
            movs       r0, #0
            bx          lr
```

```assembly
r0: value
r1: &coef[terms]
r2: x
r3: &coef[0]
r4: power
r5: coef[j]
```
Example: Polynomial evaluation

- Compiling on ARM

```c
int poly(int *coef, int terms, int x) {
    int power = 1;
    int value = 0;
    for (int j = 0; j < terms; j++) {
        value += coef[j] * power;
        power *= x;
    }
    return value;
}
```

```assembly
poly:  cmp     r1, #0
       ble     .L4
       push    {r4, r5}
       mov     r3, r0
       add     r1, r0, r1, lsl #2
       movs    r4, #1
       movs    r0, #0
       .L3:
       ldr     r5, [r3], #4
       cmp     r1, r3
       mla     r0, r4, r5, r0
       mul     r4, r2, r4
       bne     .L3
       pop     {r4, r5}
       bx      lr
       .L4:
       movs    r0, #0
       bx      lr
```

r0: value
r1: &coef[terms]
r2: x
r3: &coef[0]
r4: power
r5: coef[j]
Example: Polynomial evaluation

- Compiling on ARM

```c
for (int j = 0; j < terms; j++) {
    value += coef[j] * power;
    power *= x;
}
```

.L3:

```
ldr   r5, [r3], #4  // r5 <- coef[j]; j++  (two operations)
cmp   r1, r3        // compare: j < terms?
mla   r0, r4, r5, r0 // value += r5 * power  (mul + add)
mul   r4, r2, r4     // power *= x
bne   .L3           // repeat?
```
Example: Polynomial evaluation

- Executing \( \text{poly}(A, 3, x) \)

```assembly
cmp r1, #0
ble .L4
push {r4, r5}
mov r3, r0
add r1, r0, r1, lsl #2
movs r4, #1
movs r0, #0
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
...```

CMU 15-418/15-618, Spring 2019
Example: Polynomial evaluation

- Executing \( \text{poly}(A, 3, x) \)

```assembly
cmp r1, #0
ble .L4
push {r4, r5}
mov r3, r0
add r1, r0, r1, lsl #2
movs r4, #1
movs r0, #0
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
```

...
Example: Polynomial evaluation

- Executing `poly(A, 3, x)`

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register/Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmp</code></td>
<td>r1, #0</td>
</tr>
<tr>
<td><code>ble</code></td>
<td>.L4</td>
</tr>
<tr>
<td><code>push</code></td>
<td>{r4, r5}</td>
</tr>
<tr>
<td><code>mov</code></td>
<td>r3, r0</td>
</tr>
<tr>
<td><code>add</code></td>
<td>r1, r0, r1, lsl #2</td>
</tr>
<tr>
<td><code>movs</code></td>
<td>r4, #1</td>
</tr>
<tr>
<td><code>movs</code></td>
<td>r0, #0</td>
</tr>
<tr>
<td><code>ldr</code></td>
<td>r5, [r3], #4</td>
</tr>
<tr>
<td><code>cmp</code></td>
<td>r1, r3</td>
</tr>
<tr>
<td><code>mla</code></td>
<td>r0, r4, r5, r0</td>
</tr>
<tr>
<td><code>mul</code></td>
<td>r4, r2, r4</td>
</tr>
<tr>
<td><code>bne</code></td>
<td>.L3</td>
</tr>
<tr>
<td><code>ldr</code></td>
<td>r5, [r3], #4</td>
</tr>
<tr>
<td><code>cmp</code></td>
<td>r1, r3</td>
</tr>
<tr>
<td><code>mla</code></td>
<td>r0, r4, r5, r0</td>
</tr>
<tr>
<td><code>mul</code></td>
<td>r4, r2, r4</td>
</tr>
<tr>
<td><code>bne</code></td>
<td>.L3</td>
</tr>
<tr>
<td><code>pop</code></td>
<td>{r4, r5}</td>
</tr>
<tr>
<td><code>bx</code></td>
<td>lr</td>
</tr>
</tbody>
</table>

J=0 iteration

Preamble

J=1 iteration

J=2 iteration

Fini
Example:
Polynomial evaluation

- Executing \texttt{poly}(A, 3, x)
The software-hardware boundary

- The *instruction set architecture (ISA)* is a *functional contract* between hardware and software
  - It says *what* each instruction does, but not *how*
  - Example: Ordered sequence of x86 instructions

- A processor’s *microarchitecture* is how the ISA is implemented

Arch : \( \mu \text{Arch} \) :: Interface : Implementation
Simple CPU model

- Execute instructions in program order

- Divide instruction execution into stages, e.g.:
  1. Fetch – get the next instruction from memory
  2. Decode – figure out what to do & read inputs
  3. Execute – perform the necessary operations
  4. Commit – write the results back to registers / memory

- (Real processors have many more stages)
Evaluating polynomial on the simple CPU model

```
ldr         r5, [r3], #4
cmp         r1, r3
mla         r0, r4, r5, r0
mul         r4, r2, r4
bne         .L3

ldr         r5, [r3], #4
cmp         r1, r3
mla         r0, r4, r5, r0
mul         r4, r2, r4
bne         .L3
```

...
Evaluating polynomial on the simple CPU model

```assembly
ldr  r5, [r3], #4
cmp  r1, r3
mla  r0, r4, r5, r0
mul  r4, r2, r4
bne  .L3

ldr  r5, [r3], #4
cmp  r1, r3
mla  r0, r4, r5, r0
mul  r4, r2, r4
bne  .L3

...```

1. Read “ldr r5, [r3] #4” from memory
Evaluating polynomial on the simple CPU model

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
```

2. Decode “ldr r5, [r3] #4” and read input regs
Evaluating polynomial on the simple CPU model

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...  
```
Evaluating polynomial on the simple CPU model

4. Write values into regs r5 and r3
Evaluating polynomial on the simple CPU model

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...```
Evaluating polynomial on the simple CPU model

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
```

...
Evaluating polynomial on the simple CPU model

```
ldr  r5, [r3], #4
cmp  r1, r3
mla  r0, r4, r5, r0
mul  r4, r2, r4
bne  .L3

ldr  r5, [r3], #4
cmp  r1, r3
mla  r0, r4, r5, r0
mul  r4, r2, r4
bne  .L3

...```

CMU 15-418/15-618, Spring 2019
Evaluating polynomial on the simple CPU model

```assembly
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...```

---

CPU

Fetch | Decode | Execute | Commit

| cmp |
Evaluating polynomial on the simple CPU model

```assembly
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...```

CPU:
- Fetch
- Decode
- Execute
- Commit
Evaluating polynomial on the simple CPU model

<table>
<thead>
<tr>
<th></th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Latency = 4 ns / instr
Throughput = 1 instr / 4 ns

How fast is this processor? Latency? Throughput?
Simple CPU is very wasteful

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1dr</td>
<td>1dr</td>
<td>1dr</td>
<td>1dr</td>
</tr>
<tr>
<td></td>
<td>cmp</td>
<td>cmp</td>
<td>cmp</td>
<td>cmp</td>
</tr>
<tr>
<td></td>
<td>mla</td>
<td>mla</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hardware</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Idle</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMU 15-418/15-618, Spring 2019
Pipelining
Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr’n immediately

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
...
```
Pipelining keeps CPU busy through instruction-level parallelism

- **Idea:** Start on the next instr’n immediately

```
ldr  r5, [r3], #4
cmp  r1, r3
mla  r0, r4, r5, r0
mul  r4, r2, r4
bne  .L3

ldr  r5, [r3], #4
cmp  r1, r3
mla  r0, r4, r5, r0
mul  r4, r2, r4
bne  .L3

...```

CMU 15-418/15-618, Spring 2019
Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr’n immediately

```assembly
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

...
Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr’n immediately

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

```
Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr’n immediately

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...  
```
Pipelining keeps CPU busy through instruction-level parallelism

- **Idea:** Start on the next instr’n immediately

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...  
```
Evaluating polynomial on the pipelined CPU

<table>
<thead>
<tr>
<th></th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Latency = 4 ns / instr

How fast is this processor? Latency? Throughput?

Throughput = 1 instr / ns

4X speedup!
Speedup achieved through pipeline parallelism

Processor works on 4 instructions at a time

<table>
<thead>
<tr>
<th>Fetch</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td>cmp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMU 15-418/15-618, Spring 2019
Limitations of pipelining

- Parallelism requires **independent** work

- Q: Are instructions independent?

- A: No! Many possible *hazards* limit parallelism...
Data hazards

```
ldr ra, [rb], #4  // ra ← Memory[rb]; rb ← rb + 4
cmp rc, rd       // rc ← rd + re
```

Q: When can the CPU pipeline the `cmp` behind `ldr`?

| Fetch | ldr | cmp | ... | ... | ... | ...
|-------|-----|-----|-----|-----|-----|-----
| Decode| ldr | cmp | ... | ... | ... | ...
| Execute|    |     |     |     | ldr | cmp | ... | ...
| Commit|     |     |     |     | ldr | cmp | ... | ...

- **A**: When they use different registers
  - Specifically, when `cmp` does not read any data written by `ldr`
  - E.g., `rb != rd`
Dealing with data hazards: Stalling the pipeline

- Cannot pipeline cmp (ldr writes r3)

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```
Dealing with data hazards: Stalling the pipeline

- Cannot pipeline `cmp (ldr writes r3)`

```assembly
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

...
Dealing with data hazards: Stalling the pipeline

- Cannot pipeline `cmp (ldr writes r3)`

```
ldr   r5, [r3], #4
cmp   r1, r3
mla   r0, r4, r5, r0
mul   r4, r2, r4
bne   .L3

ldr   r5, [r3], #4
cmp   r1, r3
mla   r0, r4, r5, r0
mul   r4, r2, r4
bne   .L3
```

...
Dealing with data hazards: Stalling the pipeline

- Cannot pipeline `cmp (ldr writes r3)`

```assembly
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
```

Inject a “bubble” (NOP) into the pipeline
Dealing with data hazards: Stalling the pipeline

- Cannot pipeline `cmp` (`ldr` writes `r3`)

```
ldr  r5, [r3], #4
cmp  r1, r3
mla  r0, r4, r5, r0
mul  r4, r2, r4
bne .L3

ldr  r5, [r3], #4
cmp  r1, r3
mla  r0, r4, r5, r0
mul  r4, r2, r4
bne .L3
```

```
CPU

Fetch
mul

Decode
mla

Execute
cmp

Commit

cmp proceeds once ldr has committed
```
Stalling degrades performance

- But stalling is sometimes unavoidable
  - E.g., long-latency instructions (divide, cache miss)

<table>
<thead>
<tr>
<th>Fetch</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
</tr>
<tr>
<td>Execute</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
</tr>
<tr>
<td>Commit</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Processor works on 3 instructions at a time
Dealing with data hazards: Forwarding data

- Wait a second… data is available after Execute!

- Forwarding eliminates many (not all) pipeline stalls
Pipelining is not free!

- Q: How well does forwarding scale?
- A: Not well... many forwarding paths in deep & complex pipelines
Control hazards + Speculation

- Programs must appear to execute *in program order*
  ➔ All instructions depend on earlier ones

- Most instructions implicitly continue at the next...
- But *branches* redirect execution to new location
Dealing with control hazards: Flushing the pipeline

- What if we always fetch the next instruction?

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
pop     {r4, r5}
bx      lr
```

Static instruction sequence (i.e., program layout in memory)
Dealing with control hazards: Flushing the pipeline

- What if we always fetch the next instruction?

Static instruction sequence (i.e., program layout in memory)
Dealing with control hazards: Flushing the pipeline

- What if we always fetch the next instruction?

<table>
<thead>
<tr>
<th>ldr</th>
<th>r5, [r3], #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>r1, r3</td>
</tr>
<tr>
<td>mla</td>
<td>r0, r4, r5, r0</td>
</tr>
<tr>
<td>mul</td>
<td>r4, r2, r4</td>
</tr>
<tr>
<td>bne</td>
<td>.L3</td>
</tr>
<tr>
<td>pop</td>
<td>{r4, r5}</td>
</tr>
<tr>
<td>bx</td>
<td>lr</td>
</tr>
</tbody>
</table>

Static instruction sequence (i.e., program layout in memory)

Whoops! We fetched the wrong instructions! (Loop not finished)
Dealing with control hazards: Flushing the pipeline

What if we always fetch the next instruction?

Static instruction sequence (i.e., program layout in memory)

Whoops! We fetched the wrong instructions! (Loop not finished)
Pipeline flushes destroy performance

- Penalty increases with deeper pipelines

<table>
<thead>
<tr>
<th>Fetch</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td>cmp</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Processor works on 2 or 3 instructions at a time

Time: Processor works on 2 or 3 instructions at a time
Dealing with control hazards: *Speculation!*

- Processors do not wait for branches to execute

- Instead, they speculate (i.e., guess) where to go next + start fetching

- Modern processors use very sophisticated mechanisms
  - E.g., speculate in Fetch stage—before processor even knows instrn is a branch!
  - >95% prediction accuracy
  - Still, branch mis-speculation is major problem
Pipelining Summary

- Pipelining is a simple, effective way to improve throughput
  - $N$-stage pipeline gives up to $N \times$ speedup

- Pipelining has limits
  - Hard to keep pipeline busy because of hazards
  - Forwarding is expensive in deep pipelines
  - Pipeline flushes are expensive in deep pipelines

→ Pipelining is ubiquitous, but tops out at $N \approx 15$
Out-of-Order Execution
Increasing parallelism via dataflow

- Parallelism limited by many false dependencies, particularly sequential program order

- **Dataflow** tracks how instructions actually depend on each other
  - *True dependence*: read-after-write

*Dataflow increases parallelism by eliminating unnecessary dependences*
Example: Dataflow in polynomial evaluation

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

Loop iteration
Example: Dataflow polynomial evaluation

```assembly
ldr   r5, [r3], #4
cmp   r1, r3
mla   r0, r4, r5, r0
mul   r4, r2, r4
bne   .L3

ldr   r5, [r3], #4
cmp   r1, r3
mla   r0, r4, r5, r0
mul   r4, r2, r4
bne   .L3

...```

Loop iteration
Example: Dataflow polynomial execution

- Execution only, with perfect scheduling & unlimited execution units
  - `ldr`, `mul` execute in 2 cycles
  - `cmp`, `bne` execute in 1 cycle
  - `mla` executes in 3 cycles

- Q: Does dataflow speedup execution? By how much?

- Q: What is the performance bottleneck?
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
1. ldr
2. cmp
3. mla

lhr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
ldr r5, [r3], #4  
cmp r1, r3  
mla r0, r4, r5, r0  
mul r4, r2, r4  
bne .L3
```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
```
```assembly
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
```
Example: Dataflow polynomial execution

- Q: Does dataflow speedup execution? By how much?
  - Yes! 3 cycles / loop iteration
  - Instructions per cycle (IPC) = $5/3 \approx 1.67$
    (vs. 1 for perfect pipelining)

- Q: What is the performance bottleneck?
  - mla: Each mla depends on previous mla & takes 3 cycles
  - ➔ This program is latency-bound
Out-of-order (OoO) execution uses dataflow to increase parallelism

- Idea: Execute programs in dataflow order, but give the illusion of sequential execution

- This is a “restricted dataflow” model
  - Restricted to instructions near those currently committing
  - (Pure dataflow processors also exist that expose dataflow to software)
High-level OoO microarchitecture

Fetch → Decode → Execute → Commit

Instruction Buffer

In-order

Out-of-order

In-order
OoO is hidden behind in-order frontend & commit

- Instructions only enter & leave instruction buffer in program order; all bets are off in between!
Example: OoO polynomial evaluation

- Q: Does OoO speedup execution? By how much?

- Q: What is the performance bottleneck?

- Assume perfect forwarding & branch prediction
Example: OoO polynomial evaluation pipeline diagram
Example: OoO polynomial evaluation pipeline diagram

Fetch & Decode
   ldr    cmp

Execute
   ldr    cmp
   ldr    cmp

Commit
   ldr    cmp

TIME
Example: OoO polynomial evaluation pipeline diagram
Example: OoO polynomial evaluation pipeline diagram
Example: OoO polynomial evaluation pipeline diagram
Example: OoO polynomial evaluation pipeline diagram

<table>
<thead>
<tr>
<th>Fetch &amp; Decode</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>mul</th>
<th>bne</th>
<th>ldr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>ldr</td>
<td>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
<td>ldr</td>
<td>cmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: OoO polynomial evaluation pipeline diagram

- Wait a minute… this isn’t OoO… or even faster than a simple pipeline!
- Q: What went wrong?
- A: We’re **throughput-limited**: can only issue 1 instrn
High-level **Superscalar OoO** microarchitecture

- Must increase *pipeline width* to increase ILP > 1
Example: Superscalar OoO polynomial evaluation

```
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
```
Example: Superscalar OoO polynomial evaluation

<table>
<thead>
<tr>
<th>Fetch &amp; Decode</th>
<th>ldr</th>
<th>mla</th>
<th>bne</th>
<th>cmp</th>
<th>mul</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cmp</td>
<td>mul</td>
<td>ldr</td>
<td>mla</td>
<td>bne</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execute</th>
</tr>
</thead>
</table>

| Commit |

```
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
```
Example: Superscalar OoO polynomial evaluation

<table>
<thead>
<tr>
<th>Fetch &amp; Decode</th>
<th>ldr</th>
<th>mla</th>
<th>bne</th>
<th>cmp</th>
<th>mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>mul</td>
<td>ldr</td>
<td>mla</td>
<td>bne</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execute</th>
<th>ldr</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Commit</th>
</tr>
</thead>
</table>

```
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
```
Example: Superscalar OoO polynomial evaluation

<table>
<thead>
<tr>
<th>Fetch &amp; Decode</th>
<th>ldr</th>
<th>mla</th>
<th>bne</th>
<th>cmp</th>
<th>mul</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cmp</td>
<td>mul</td>
<td>ldr</td>
<td>mla</td>
<td>bne</td>
</tr>
<tr>
<td>Execute</td>
<td>ldr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mul</td>
</tr>
</tbody>
</table>

TIME

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```
Example: Superscalar OoO polynomial evaluation

Fetch & Decode
- ldr
- mla
- bne
- cmp
- mul
- cmp
- mul
- ldr
- mla
- bne

Execute
- ldr
- cmp
- mla
- mul
- bne
- ldr
- cmp
- mla
- mul
- bne

Commit

TIME

ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
ldr r5, [r3], #4
ncmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
Example: Superscalar OoO polynomial evaluation

Fetch & Decode:
- ldr
- mla
- bne
- cmp
- mul

Execute:
- cmp
- mul
- ldr
- mla
- bne
- ldr
- cmp
- bne
- mla
- mul

Commit:
- mul
- ldr

TIME

ldr  r5, [r3], #4

cmp  r1, r3

mla  r0, r4, r5, r0

mul  r4, r2, r4

bne  .L3

ldr  r5, [r3], #4

cmp  r1, r3

mla  r0, r4, r5, r0

mul  r4, r2, r4

bne  .L3

CMU 15-418/15-618, Spring 2019
Example: Superscalar OoO polynomial evaluation

<table>
<thead>
<tr>
<th>Fetch &amp; Decode</th>
<th>ldr</th>
<th>mla</th>
<th>bne</th>
<th>cmp</th>
<th>mul</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cmp</td>
<td>mul</td>
<td>ldr</td>
<td>mla</td>
<td>bne</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execute</th>
<th>ldr</th>
<th>cmp</th>
<th>bne</th>
<th>mul</th>
<th>mla</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mla</td>
<td>mla</td>
<td>ldr</td>
<td>cmp</td>
<td>bne</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Commit</th>
<th>mul</th>
<th>ldr</th>
<th>cmp</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```
Example: Superscalar OoO polynomial evaluation

TIME

Fetch & Decode
<table>
<thead>
<tr>
<th>ldr</th>
<th>mla</th>
<th>bne</th>
<th>cmp</th>
<th>mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>mul</td>
<td>ldr</td>
<td>mla</td>
<td>bne</td>
</tr>
</tbody>
</table>

Execute
<table>
<thead>
<tr>
<th>ldr</th>
<th>cmp</th>
<th>bne</th>
<th>mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>mla</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Commit
<table>
<thead>
<tr>
<th>ldr</th>
<th>cmp</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>mla</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
ldr r5, [r3], #4
mla r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3
```
Example: Superscalar OoO polynomial evaluation

Observe:
- Front-end & commit in-order (i.e., left-to-right)
- Execute out-of-order
Example: Superscalar OoO polynomial evaluation

Fetch & Decode

<table>
<thead>
<tr>
<th></th>
<th>ldr</th>
<th>mla</th>
<th>bne</th>
<th>cmp</th>
<th>mul</th>
<th>ldr</th>
<th>mla</th>
<th>bne</th>
<th>cmp</th>
<th>mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>mul</td>
<td>ldr</td>
<td>mla</td>
<td>bne</td>
<td>cmp</td>
<td>mul</td>
<td>ldr</td>
<td>mla</td>
<td>bne</td>
<td>cmp</td>
</tr>
</tbody>
</table>

Execute

<table>
<thead>
<tr>
<th></th>
<th>ldr</th>
<th>cmp</th>
<th>bne</th>
<th>mul</th>
<th>ldr</th>
<th>cmp</th>
<th>bne</th>
<th>mul</th>
<th>ldr</th>
<th>cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>mla</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mla</td>
<td></td>
<td></td>
<td></td>
<td>mla</td>
<td></td>
</tr>
<tr>
<td>mla</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mla</td>
<td></td>
<td></td>
<td></td>
<td>mla</td>
<td></td>
</tr>
<tr>
<td>mla</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mla</td>
<td></td>
<td></td>
<td></td>
<td>mla</td>
<td></td>
</tr>
</tbody>
</table>

Commit

<table>
<thead>
<tr>
<th></th>
<th>ldr</th>
<th>cmp</th>
<th>mla</th>
<th>bne</th>
<th>cmp</th>
<th>mla</th>
<th>bne</th>
<th>cmp</th>
<th>mla</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>mul</td>
<td>ldr</td>
<td>cmp</td>
<td>bne</td>
<td>mul</td>
<td>ldr</td>
<td>cmp</td>
<td>bne</td>
<td>mul</td>
<td>ldr</td>
<td>cmp</td>
</tr>
</tbody>
</table>

One loop iteration / 3 cycles!
Structural hazards: Other throughput limitations

- Execution units are specialized
  - Floating-point (add/multiply)
  - Integer (add/multiply/compare)
  - Memory (load/store)

- Processor designers must choose which execution units to include and how many

- Structural hazard: Data is ready, but instr’n cannot issue because no hardware is available
Example: Structural hazards can severely limit performance

One loop iteration / 5 cycles 😞
Superscalar scheduling is complex & hard to scale

- Q: When is it safe to issue two instructions?
- A: When they are independent
  - Must compare all pairs of input and output registers

- Scalability: $O(W^2)$ comparisons where $W$ is issue width
OoO x86: Microcoding

- Each x86 instruction describes several operations
  - E.g., `add [esp+4], 5` means:
    1. Load `Mem[esp+4]`
    2. Add 5 to it
    3. Store result to `Mem[esp+4]`

- This is too much for (fast) hardware

- Instead, hardware decodes instr’ns into *micro-ops*
  - Rest of pipeline uses micro-ops
...But wait, there’s more!

- Many issues we could not touch on
- How to eliminate false dependences
  - E.g., write-after-read / write-after-write
- How to track dependences through memory
  - E.g., store→load forwarding
- How to rollback mis-speculations
Recall from last time: ILP tapped out... why?

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

Processor clock rate stops increasing

No further benefit from ILP

CMU 15-418/15-618, Spring 2019
Limitations of ILP

- ILP works great! ...But is complex + hard to scale
- 4-wide superscalar × 20-stage pipeline = 80 instrns in flight
- High-performance OoO buffers hundreds of instructions

- Pipelines can only go so deep
  - Branch misprediction penalty grows
  - Frequency (GHz) limited by power
- Programs have limited ILP
  - Even with perfect scheduling, >8-wide issue doesn’t help
- Dynamic scheduling overheads are significant
- Out-of-order scheduling is expensive
Limitations of ILP ➞ Multicore

- ILP works great! ...But is complex + hard to scale

- From hardware perspective, multicore is much more efficient, but...

- Parallel software is hard!
  - Industry resisted multicore for as long as possible
  - When multicore finally happened, CPU μarch simplified ➞ more cores
  - Many program(mer)s still struggle to use multicore effectively