Lecture 2:

Instruction-Level Parallelism

15-418 Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2020
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 ➔ Few cores (tens)
  - (Relatively) Easy to write fast software

- GPUs: Software makes parallelism explicit
  - Simple, cheap hardware ➔ 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 sequential 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
```
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
```

r0: value
r1: &coef[terms]
r2: x
r3: &coef[j]
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
    ...
```
Example:
Polynomial evaluation

- Executing poly(A, 3, x)

```
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 \( poly(A, 3, x) \)

```assembly
Preamble
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

J=0 iteration
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

J=1 iteration
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

J=2 iteration
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

Fini
pop    {r4, r5}
bx     lr
```
Example: Polynomial evaluation

- Executing poly(A, 3, x)

```
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
...  
```

Preamble

```
J=0 iteration
```

```
J=1 iteration
```

```
J=2 iteration
```

```
Fini
```

...
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 : \( \mu \text{Arch} :: \text{Interface} : \text{Implementation} \)
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

...    
```

CPU

- Fetch
- Decode
- Execute
- Commit
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
```

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

3. Load memory at r3 and compute r3 + 4
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

...```

4. Write values into regs r5 and r3
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
```

...
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
...
```
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

```
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>Fetch</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>
</tr>
<tr>
<td>Execute</td>
<td>ldr</td>
<td>cmp</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>ldr</td>
<td>cmp</td>
<td></td>
</tr>
</tbody>
</table>

How fast is this processor?  
Latency?  Throughput?

Latency = 4 ns / instr  
Throughput = 1 instr / 4 ns
Simple CPU is very wasteful

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

...```

CPU

```
Fetch
  ldr
Decode
Execute
  ldr
Commit```

CMU 15-418/15-618, Spring 2019
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

```
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

```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 pipelined CPU

How fast is this processor? Latency? Throughput?

<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></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>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></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>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></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>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>

Latency = 4 ns / instr

Throughput = 1 instr / ns

4X speedup!
**Speedup achieved through pipeline parallelism**

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

Processor works on 4 instructions at a time

---

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

```assembly
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
```

CPU

Fetch

Decode

Execute

Commit
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

...```

CPU

- Fetch
- Decode
  - `cmp ??`
  - `ldr`
- Execute
- Commit
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)`

```
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
```

...
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>cmp</td>
<td>mla</td>
<td>mul</td>
<td>bne</td>
</tr>
</tbody>
</table>

Processor works on 3 instructions at a time

TIME

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
### Speedup achieved through pipeline parallelism

Processor works on 4 instructions at a time 😊

<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>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>cmp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMU 15-418/15-618, Spring 2019
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?

```
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?

```
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)

CPU

Fetch
Decode
Execute
Commit

bx
pop
bne
mul

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

- What if we always fetch the next instruction?

```assembly
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)

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

- Penalty increases with deeper pipelines

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

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 $N \approx 15$
Software Takeaways

- Processors with a simple “in-order” pipeline are very sensitive to running “good code”
  - Compiler should target a specific model of CPU
  - Low-level assembly hacking

- But very few CPUs are in-order these days
  - E.g., embedded, ultra-low-power applications

- Instead, ≈ all modern CPUs are “out-of-order”
  - Even in classic “low-power domains” (like mobile)
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

```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 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: Dataflow polynomial execution

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

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

- Q: What is the performance bottleneck?
1  ldr
12  ldr   r5, [r3], #4
13  cmp   r1, r3
14  mla   r0, r4, r5, r0
15  mul   r4, r2, r4
16  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
```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3
ldr

cmp

mla

mul

bne

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

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
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
Latency Bound

- What is the “critical path” of the computation?
  - Longest path across iterations in dataflow graph
  - E.g., mla in last slide (but could be multiple ops)

- Critical path limits maximum performance

- Real CPUs may not achieve latency bound, but useful mental model + tool for program analysis
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
Example: OoO polynomial evaluation pipeline diagram

Fetch & Decode: ldr, cmp, mla

Execute: ldr, cmp, mla

Commit: ldr, cmp, mla
Example: OoO polynomial evaluation pipeline diagram

Fetch & Decode
- ldr
- cmp
- mla
- mul

Execute
- ldr
- cmp
- mla
- mul

Commit
- ldr
- cmp
- mla
- mul
Example: OoO polynomial evaluation pipeline diagram

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

Execute
- ldr
- cmp
- mla
- mul
- bne

Commit
- ldr
- cmp
- mla
- mul
- bne
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>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>
</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>
<td></td>
</tr>
<tr>
<td></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>
</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 exec 1 instrn
High-level **Superscalar OoO** microarchitecture

- Must increase *pipeline width* to increase ILP > 1

In-order

Out-of-order

In-order

CMU 15-418/15-618, Spring 2019
Focus on Execution, not Fetch & Commit

- Goal of OoO design is to only be limited by dataflow execution
- Fetch and commit are over-provisioned so that they (usually) do not limit performance
  ➔ Programmers can (usually) ignore fetch/commit

- **Big Caveat:** Programs with *inherently unpredictable* control flow will often be limited by fetch stalls (branch misprediction)
  - E.g., branching based on random data
Example: Superscalar OoO polynomial evaluation

Fetch & Decode
- ldr
- cmp

Execute

Commit

```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: 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>

```
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>ml</th>
<th>bne</th>
<th>cmp</th>
<th>mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm</td>
<td>cm</td>
<td>ml</td>
<td>ldr</td>
<td>ml</td>
<td>bne</td>
</tr>
<tr>
<td>ldr</td>
<td>r5</td>
<td>[r3], #4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td>r1, r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ml</td>
<td>r0, r4, r5, r0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>r4, r2, r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>.L3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldr</td>
<td>r5</td>
<td>[r3], #4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td>r1, r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ml</td>
<td>r0, r4, r5, r0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>r4, r2, r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>.L3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Superscalar OoO polynomial evaluation

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

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

Commit:
- ldr
- cmp
- mla
- mul
- bne
- ldr
- cmp
- mla
- mul
- bne

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

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

Execute
  ldr  cmp
  mla
  mul

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

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

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

Commit:
- ldr, cmp, bne
- mla
- mul
- bne

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

Commit
- ldr
- cmp
- bne
- mla
- mul
- bne

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  bne  mul

mla

mul

ldr  cmp  bne

mla

mul

ldr  cmp  bne

mul

mul

Commit
ldr  cmp  mla  bne  cmp  mla  bne

mul  ldr

mul

TIME

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

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

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

| Mem Execute   | ldr | ldr | ldr | ldr | ldr | ldr | ldr | ldr | ldr | ldr | ldr | ldr | ldr | ldr | ldr | ldr |

| Int Execute   | cmp | bne | cmp | bne | cmp | bne | cmp | bne | cmp | bne | cmp | bne | cmp | bne | cmp | bne |

| Mult Execute  | mla | mul | mla | mul | mla | mul | mla | mul | mla | mul | mla | mul | mla | mul | mla | mul |

| Commit        | ldr | cmp | mla | mul | ldr | mla | mul | ldr | mla | mul | ldr | mla | mul | ldr | mla | mul |

|               | bne | cmp | bne | cmp | bne | cmp | bne | cmp | bne | cmp | bne | cmp | bne | cmp | bne | cmp |

One loop iteration / 5 cycles 😞
Throughput Bound

- **Ingredients:**
  - Number of operations to perform (of each type)
  - Number & issue rate of “execution ports”/“functional units” (of each type)

- **Throughput bound = ops / issue rate**
  - E.g., \((1 \text{ mla} + 1 \text{ mul}) / (2 + 3 \text{ cycles})\)

- Again, a real CPU might not exactly meet this bound
Software Takeaway

- OoO is much less sensitive to “good code”
  - Better performance portability
  - Of course, compiler still matters

- OoO makes performance analysis much simpler
  - **Throughput bound**: Availability of execution ports
  - **Latency bound**: “Critical path” latency
  - Slowest gives good approximation of program perf
Out-of-Order Execution: Under the Hood
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
Register Renaming

- “False dependences” can severely limit parallelism
  - Write-after-read
  - Write-after-write

- OoO processors eliminate false dependences by transparently renaming registers
  - CPU has many more “physical” than “architectural” registers
  - Each time register is written, it is allocated to a new physical register
  - Physical registers freed when instructions commit
Memory Disambiguation

- CPU must respect store → load ordering
  - E.g., a later instruction reads a value from memory written by an earlier instruction

- But what if the OoO CPU executes the load first?
  - Must “rollback” + execute the load again (next slide)

- Corollary: OoO CPU must track the order of all loads & stores, and only write memory when a store commits
Rollback & Recovery

- OoO CPUs speculate constantly to improve performance
  - E.g., even guessing the results of a computation (“value prediction”)

- Need mechanisms to “rollback” to an earlier point in execution when speculation goes awry
  - Complex: Need to recover old register names, flush pending memory operations, etc

- Very expensive: Up to hundreds of instrns of work lost!
Scaling Instruction-Level Parallelism
Recall from last time:
ILP & pipelining tapped out... why?
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” of processor
  - Not great!
Limitations of ILP

- 4-wide superscalar $\times$ 20-stage pipeline $= 80$ instrns in flight
- High-performance OoO buffers *hundreds* of instructions

- **Programs have limited ILP**
  - Even with perfect scheduling, $>8$-wide issue doesn’t help

- Pipelines can only go so deep
  - Branch misprediction penalty grows
  - Frequency (GHz) limited by power

- 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