Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two LEGv8 implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: LDUR, STUR
  - Arithmetic/logical: add, sub, and, or, slt
  - Control transfer: beq, j
Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address or PC + 4

CPU Overview
### Multiplexers

- Can’t just join wires together
- Use multiplexers

### Control

[Image of a control diagram]
Logic Design Basics

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

- **AND-gate**
  - \( Y = A \& B \)

- **Multiplexer**
  - \( Y = S ? I_1 : I_0 \)

- **Adder**
  - \( Y = A + B \)

- **Arithmetic/Logic Unit**
  - \( Y = F(A, B) \)
Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

![Diagram of a register with inputs Clk and D, and output Q.](image1)

Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

![Diagram of a register with inputs Clk, Write, and D, and output Q.](image2)
Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux’s, memories, …
  - We will build a LEGv8 datapath incrementally
    - Refining the overview design
Instruction Fetch

- 32-bit register
  - PC
  - Read address
  - Instruction
  - Instruction memory
  - Increment by 4 for next instruction

R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

a. Registers

b. ALU
Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch
Branch Instructions

Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions
R-Type/Load/Store Datapath

Full Datapath
ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode

<table>
<thead>
<tr>
<th>ALU control</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>pass input b</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>

Assume 2-bit ALUOp derived from opcode
- Combinational logic derives ALU control

<table>
<thead>
<tr>
<th>opcode</th>
<th>ALUOp</th>
<th>Operation</th>
<th>Opcode field</th>
<th>ALU function</th>
<th>ALU control</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDUR</td>
<td>00</td>
<td>load register</td>
<td>XXXXXXXXXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>STUR</td>
<td>00</td>
<td>store register</td>
<td>XXXXXXXXXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>CBZ</td>
<td>01</td>
<td>compare and branch on zero</td>
<td>XXXXXXXXXXXXXX</td>
<td>pass input b</td>
<td>0111</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AND</td>
<td>100100</td>
<td>AND</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
<td>100101</td>
<td>OR</td>
<td>0001</td>
</tr>
</tbody>
</table>
The Main Control Unit

Control signals derived from instruction

<table>
<thead>
<tr>
<th>Field</th>
<th>opcode</th>
<th>Rm</th>
<th>shamt</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit positions</td>
<td>31:21</td>
<td>20:16</td>
<td>15:10</td>
<td>9:5</td>
<td>4:0</td>
</tr>
</tbody>
</table>

a. R-type instruction

<table>
<thead>
<tr>
<th>Field</th>
<th>1986 or 1984</th>
<th>address</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit positions</td>
<td>31:21</td>
<td>20:12</td>
<td>11:10</td>
<td>9:5</td>
<td>4:0</td>
</tr>
</tbody>
</table>

b. Load or store instruction

<table>
<thead>
<tr>
<th>Field</th>
<th>180</th>
<th>address</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit positions</td>
<td>31:26</td>
<td>23:5</td>
<td>4:0</td>
</tr>
</tbody>
</table>

c. Conditional branch instruction

Datapath With Control

Chapter 4 — The Processor — 23

Chapter 4 — The Processor — 24
R-Type Instruction

Load Instruction
CBZ Instruction

Implementing Uncnd’l Branch

Jump uses word address

Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00

Need an extra control signal decoded from opcode
**Datapath With B Added**

**Performance Issues**

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining
Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

- Four loads:
  - Speedup
    \[ \frac{8}{3.5} = 2.3 \]

- Non-stop:
  - Speedup
    \[ \frac{2n}{0.5n + 1.5} \approx 4 \]
    \[ = \text{number of stages} \]

LEGv8 Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDUR</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>STUR</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>CBZ</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

Chapter 4 — The Processor — 33

Pipeline Performance

Program execution order (in instructions)

Single-cycle \((T_c = 800\text{ps})\)

Pipelined \((T_c = 200\text{ps})\)
PipeLine Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions_{pipelined} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- LEGv8 ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle
Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource
- In LEGv8 pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to stall for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - ADD \( x_{19}, x_0, x_1 \)
  - SUB \( x_2, x_{19}, x_3 \)

Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F;

```
LDUR X1, [X0,#0]
LDUR X2, [X0,#8]
ADD X3, X1, X2
STUR X3, [X0,#24]
LDUR X4, [X0,#16]
ADD X5, X1, X4
STUR X5, [X0,#32]
```

13 cycles

```
LDUR X1, [X0,#0]
LDUR X2, [X0,#8]
LDUR X4, [X0,#16]
ADD X3, X1, X2
STUR X3, [X0,#24]
ADD X5, X1, X4
STUR X5, [X0,#32]
```

11 cycles
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

- In LEGv8 pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction
Branch Prediction

- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In LEGv8 pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

LEGv8 Pipelined Datapath

Right-to-left flow leads to hazards

Chapter 4 — The Processor — 47
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle

Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
  - We’ll look at “single-clock-cycle” diagrams for load & store
Chapter 4 — The Processor — 51

IF for Load, Store, ...

Chapter 4 — The Processor — 52

ID for Load, Store, ...
WB for Load

Corrected Datapath for Load
WB for Store

Multi-Cycle Pipeline Diagram

- Form showing resource usage

Program execution order (in instructions)

LDUR X10, [X1,#40] IM Reg ALU DM Reg

SUB X11, X2, X3 IM Reg ALU DM Reg

ADD X12, X3, X4 IM Reg ALU DM Reg

LDUR X13, [X1,#48] IM Reg ALU DM Reg

ADD X14, X5, X6 IM Reg ALU DM Reg
Multi-Cycle Pipeline Diagram

- Traditional form

```
<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>order (in instructions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDUR X10, [X1, #40]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB X11, X2, X3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD X12, X3, X4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDUR X13, [X1, #48]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD X14, X5, X6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle
Pipelined Control (Simplified)

Control signals derived from instruction
- As in single-cycle implementation
Data Hazards in ALU Instructions

Consider this sequence:

- SUB \( x_2, x_1, x_3 \)
- AND \( x_{12}, x_2, x_5 \)
- OR \( x_{13}, x_6, x_2 \)
- ADD \( x_{14}, x_2, x_2 \)
- STUR \( x_{15}, [x_2, #100] \)

- We can resolve hazards with forwarding
  - How do we detect when to forward?
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
  - ALU operand register numbers in EX stage are given by
    - ID/EX.RegisterRn1, ID/EX.RegisterRm2
- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRn1
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRm2
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRn1
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRm2
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite

- And only if Rd for that instruction is not XZR
  - EX/MEM.RegisterRd ≠ 31, MEM/WB.RegisterRd ≠ 31

Forwarding Paths
## Forwarding Conditions

<table>
<thead>
<tr>
<th>Mux control</th>
<th>Source</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForwardA = 00</td>
<td>ID/EX</td>
<td>The first ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardA = 10</td>
<td>EX/MEM</td>
<td>The first ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardA = 01</td>
<td>MEM/WB</td>
<td>The first ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
<tr>
<td>ForwardB = 00</td>
<td>ID/EX</td>
<td>The second ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardB = 10</td>
<td>EX/MEM</td>
<td>The second ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardB = 01</td>
<td>MEM/WB</td>
<td>The second ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
</tbody>
</table>

## Double Data Hazard

- Consider the sequence:
  
  \[
  \begin{align*}
  \text{add} & \quad x_1, x_1, x_2 \\
  \text{add} & \quad x_1, x_1, x_3 \\
  \text{add} & \quad x_1, x_1, x_4 \\
  \end{align*}
  \]

- Both hazards occur
  - Want to use the most recent
  - Revise MEM hazard condition
    - Only fwd if EX hazard condition isn’t true
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 31) and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31) and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRn1)) and (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 31) and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31) and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRm2)) and (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01

Datapath with Forwarding
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRn1, IF/ID.RegisterRm2
- Load-use hazard when
  - ID/EX.MemRead and
    - ((ID/EX.RegisterRd = IF/ID.RegisterRn1) or
      - (ID/EX.RegisterRd = IF/ID.RegisterRm1))
  - If detected, stall and insert bubble

How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for LDUI
    - Can subsequently forward to EX stage
Load-Use Data Hazard

Datapath with Hazard Detection
Stalls and Performance

**The BIG Picture**

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

---

Branch Hazards

- If branch outcome determined in MEM

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program execution order (in instructions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40 CBZ X1, 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44 AND X12, X2, X5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48 ORR X13, X6, X2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52 ADD X14, X2, X2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72 LDUR X4, [X7,#100]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flush these instructions (Set control values to 0)
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken
  36:  SUB  X10, X4, X8
  40:  CBZ  X1,  X3,  8
  44:  AND  X12, X2, X5
  48:  ORR  X13, X2, X6
  52:  ADD  X14, X4, X2
  56:  SUB  X15, X6, X7
  ...
  72:  LDUR  X4, [X7,#50]

Example: Branch Taken
Example: Branch Taken

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction
1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

  outer: ...
  ...
  inner: ...
  ...
  CBZ ..., ..., inner
  ...
  CBZ ..., ..., outer

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions
Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard
Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In LEGv8: Exception Link Register (ELR)

- Save indication of the problem
  - In LEGv8: Exception Syndrome Register (ESR)
  - We’ll assume 1-bit
    - 0 for undefined opcode, 1 for overflow

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause

- Exception vector address to be added to a vector table base register:
  - Unknown Reason: $00\,0000_{\text{two}}$
  - Overflow: $10\,1100_{\text{two}}$
  - …: $11\,1111_{\text{two}}$

- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, ...

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  \[ \text{ADD } X1, X2, X1 \]
  - Prevent X1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set ESR and ELR register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware
Pipeline with Exceptions

Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch

- PC saved in ELR register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust
Exception Example

- Exception on ADD in
  40 SUB  X11, X2, X4
  44 AND X12, X2, X5
  48 ORR X13, X2, X6
  4C ADD X1, X2, X1
  50 SUB X15, X6, X7
  54 LDUR X16, [X7,#100]
  ...

- Handler
  80000180 STUR X26, [X0,#1000]
  80000184 STUR X27, [X0,#1008]
  ...

Chapter 4 — The Processor — 95

Exception Example

Chapter 4 — The Processor — 96
Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!
Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice
Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

- Dynamic multiple issue
  - 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

Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing

- Common to static and dynamic multiple issue

- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated
Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)
Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary
LEGv8 with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Hazards in the Dual-Issue LEGv8

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet
    - ADD X0, X0, X1
    - LDUR X2, [X0,#0]
  - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

Schedule this for dual-issue LEGv8

Loop: LDUR X0, [X20,#0] // X0=array element
ADD X0, X0,X21 // add scalar in X21
STUR X0, [X20,#0] // store result
SUBI X20, X20,#4 // decrement pointer
CMP X20, X22 // branch $s1!=0
BGT Loop

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: nop</td>
<td>LDUR X0, [X20,#0]</td>
<td>1</td>
</tr>
<tr>
<td>SUBI X20, X20,#4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>ADD X0, X0,X21</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>CMP X20, X22</td>
<td>sw $t0, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>BGT Loop</td>
<td>STUR X0, [X20,#0]</td>
<td>5</td>
</tr>
</tbody>
</table>

- IPC = 7/6 = 1.17 (c.f. peak IPC = 2)
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 Example

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>SUBI X20, X20,#32</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>ADD X0, X0, X21</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>ADD X1, X1, X21</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>ADD X2, X2, X21</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>ADD X3, X3, X21</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>CMP X20,X22</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>BGT Loop</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>LDUR X0, [X20,#0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDUR X1, [X20,#24]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDUR X2, [X20,#16]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDUR X3, [X20,#8]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STUR X0, [X20,#32]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw X1, [X20,#24]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw X2, [X20,#16]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw X3, [X20,#8]</td>
<td></td>
</tr>
</tbody>
</table>

- IPC = \(15/8 = 1.875\)
  - Closer to 2, but at cost of registers and code size
Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example
  
  ```
  LDUR  X0, [X21,#20]
  ADD  X1, X0, X2
  SUB  X23,X23,X3
  ANDI X5, X23,#20
  ```
  - Can start sub while ADD is waiting for LDUI
Dynamically Scheduled CPU

Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
    - Register update may not be required
Speculation

- Predict branch and continue issuing
  - Don’t commit until branch outcome determined
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don’t commit load until speculation cleared

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predictable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards
Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>UltraSparc III</td>
<td>2003</td>
<td>1950MHz</td>
<td>14</td>
<td>4</td>
<td>No</td>
<td>1</td>
<td>90W</td>
</tr>
<tr>
<td>UltraSparc T1</td>
<td>2005</td>
<td>1200MHz</td>
<td>6</td>
<td>1</td>
<td>No</td>
<td>8</td>
<td>70W</td>
</tr>
</tbody>
</table>
## Cortex A53 and Intel i7

<table>
<thead>
<tr>
<th>Processor</th>
<th>ARM A53</th>
<th>Intel Core i7 920</th>
</tr>
</thead>
<tbody>
<tr>
<td>Market</td>
<td>Personal Mobile Device</td>
<td>Server, cloud</td>
</tr>
<tr>
<td>Thermal design power</td>
<td>100 milliWatts (1 core @ 1 GHz)</td>
<td>130 Watts</td>
</tr>
<tr>
<td>Clock rate</td>
<td>1.5 GHz</td>
<td>2.66 GHz</td>
</tr>
<tr>
<td>Cores/Chip</td>
<td>4 (configurable)</td>
<td>4</td>
</tr>
<tr>
<td>Floating point?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiple issue?</td>
<td>Dynamic</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Peak instructions/clock cycle</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Pipeline stages</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>Pipeline schedule</td>
<td>Static in-order</td>
<td>Dynamic out-of-order with speculation</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>Hybrid</td>
<td>2-level</td>
</tr>
<tr>
<td>1st level caches/core</td>
<td>16-64 KiB I, 16-64 KiB D</td>
<td>32 KiB I, 32 KiB D</td>
</tr>
<tr>
<td>2nd level caches/core</td>
<td>128-2048 KiB</td>
<td>256 KiB (per core)</td>
</tr>
<tr>
<td>3rd level caches (shared)</td>
<td>(platform dependent)</td>
<td>2-8 MB</td>
</tr>
</tbody>
</table>

### ARM Cortex-A53 Pipeline

The ARM Cortex-A53 pipeline is designed for efficiency and performance. It includes stages for instruction fetch, decode, execute, and writeback. The pipeline is optimized for both integer and floating-point operations, with separate execution units for ALU and MAC (Multiply-Accumulate) instructions. The pipeline stages are labeled D1, D2, and D3, with additional stages for floating-point operations.
ARM Cortex-A53 Performance

Chapter 4 — The Processor — 123

Core i7 Pipeline

Chapter 4 — The Processor — 124
Matrix Multiply

Unrolled C code

```c
#include <x86intrin.h>
#define UNROLL (4)

void dgemm(int n, double* A, double* B, double* C)
{
    for (int i = 0; i < n; i += UNROLL*4)
        for (int j = 0; j < n; j++)
            __m256d c[4];

    for (int x = 0; x < UNROLL; x++)
        c[x] = _mm256_load_pd(C+i+x*4+j*n);

    for (int k = 0; k < n; k++)
        __m256d b = _mm256_broadcast_sd(B+k+j*n);

    for (int x = 0; x < UNROLL; x++)
        c[x] = _mm256_add_pd(c[x],
                          _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));

    for (int x = 0; x < UNROLL; x++)
        _mm256_store_pd(C+i+x*4+j*n, c[x]);
}
```
Matrix Multiply

Assembly code:

1. `vmovapd (%r11),%ymm4`  # Load 4 elements of C into %ymm4
2. `mov %rbx,%rax`  # register %rax = %rbx
3. `xor %ecx,%ecx`  # register %ecx = 0
4. `vmovapd 0x20(%r11),%ymm3`  # Load 4 elements of C into %ymm3
5. `vmovapd 0x40(%r11),%ymm2`  # Load 4 elements of C into %ymm2
6. `vmovapd 0x60(%r11),%ymm1`  # Load 4 elements of C into %ymm1
7. `vbroadcastsd (%rcx,%r9,1),%ymm0`  # Make 4 copies of B element
8. `add $0x8,%rcx`  # register %rcx = %rcx + 8
9. `vmulpd (%rax),%ymm0,%ymm5`  # Parallel mul %ymm1, 4 A elements
10. `vaddpd %ymm5,%ymm4,%ymm4`  # Parallel add %ymm5, %ymm4
11. `vmulpd 0x20(%rax),%ymm0,%ymm5`  # Parallel mul %ymm1, 4 A elements
12. `vaddpd %ymm5,%ymm3,%ymm3`  # Parallel add %ymm5, %ymm3
13. `vmulpd 0x40(%rax),%ymm0,%ymm5`  # Parallel mul %ymm1, 4 A elements
14. `vaddpd %ymm5,%ymm2,%ymm2`  # Parallel add %ymm5, %ymm2
15. `vaddpd %ymm0,%ymm1,%ymm1`  # Parallel add %ymm0, %ymm1
16. `cmp %r8,%rcx`  # compare %r8 to %rcx
17. `vaddpd %ymm5,%ymm2,%ymm2`  # Parallel add %ymm5, %ymm2
18. `vaddpd %ymm5,%ymm1,%ymm1`  # Parallel add %ymm5, %ymm1
19. `jne <dgemm+0x68>`  # jump if not %r8 != %rax
20. `add $0x1,%esi`  # register %esi = %esi + 1
21. `vmovapd %ymm4,(%r11)`  # Store %ymm4 into 4 C elements
22. `vmovapd %ymm3,0x20(%r11)`  # Store %ymm3 into 4 C elements
23. `vmovapd %ymm2,0x40(%r11)`  # Store %ymm2 into 4 C elements
24. `vmovapd %ymm1,0x60(%r11)`  # Store %ymm1 into 4 C elements

Performance Impact

![Graph showing performance impact](image-url)
Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven’t we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots
Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall