ELEC6234
Embedded Processor Design

Pipelined CPUs
Key features of RISC and pipelines

– Small and simple instruction set
– Most instructions execute in single clock cycle (no microcode)
– Memory access instructions only transfer data: memory <-> registers
– ALU operations are register to register
– Large number of general purpose registers

– RISC lends itself easily to efficient instruction execution pipeline
  • Pipelines accelerate performance

– Pipelines are not always used in embedded architectures
  • Hardware overheads are necessary to implement and control a pipeline
  • But many embedded processors are pipelined, eg. ARM Cortex cores
    – ARM Cortex M3 – 3 stage pipeline
    – ARM Cortex A9 – dynamic length, 9-11 stage pipeline
Even simplest microcontrollers use pipelines

- AVR has a two-stage pipeline: parallel instruction fetches and instruction executions.
- This is enabled by the Harvard architecture of the AVR and the use of Instruction Register.
- This basic pipelining results in fast execution, single-clock cycle per instruction.

**Figure 6. The Parallel Instruction Fetches and Instruction Executions**

AVR parallel fetch and execution

- Instruction loaded into Instruction Register executes
- At the same time next instruction is fetched from Program Memory
Pipeline hazards

- Instruction cycle broken into \( n \) phases (one execution stage per phase)
  - e.g. Fetch, Decode, ReadOPs, Execute1, Execute2, WriteBack
- A new instruction is fetched at each phase
- Maximum speed gain is \( n \) times
- Pipeline hazards reduce the ability to achieve a gain of \( n \) times
  - Types of Hazards
    - Resource
      - Hazard occurs when instruction needs a resource being used by another instruction
    - Data
      - RAW (read-after-write hazard: read is requested before write has finished)
      - WAR (write-after-read hazard: write is requested before read is finished)
      - WAW (write-after-write hazard: writes occur in an unintended order)
    - Control
      - Hazard occurs when a wrong fetch decision at a branch results in an extra instruction fetch and a pipeline flush
Data hazards

• Dependences between instructions may cause data hazards when $Instr_1$ and $Instr_2$ are so close that their overlapping within the pipeline would change their access order to $Reg$.

• Three types of data hazards:

  Read After Write (RAW): $Instr_2$ tries to read operand before $Instr_1$ writes it.

  Write After Read (WAR): $Instr_2$ tries to write operand before $Instr_1$ reads it.

  Write After Write (WAW): $Instr_2$ tries to write operand before $Instr_1$ writes it.
Data access hazard - example

load Reg1,A
load Reg2,B
add Reg2,Reg1,Reg2
mul Reg1,Reg2,Reg1
Pipeline conflict due to data dependency hazard

```
add Reg2,Reg1,Reg2
mul Reg1,Reg2,Reg1
```

wrong register read!

```
IF  ID  EX  MEM  WB
```

```
IF  ID  EX  MEM  WB
```

cycle time  time
Solutions to data access and data dependency hazards

- **Software solutions:** (Compiler scheduling):
  - Putting no-op instructions after each instruction that may cause a hazard
  - Instruction scheduling: rearrange code to reduce no-ops

- **Hardware solutions:**
  - Hazard detection hardware is necessary.
  - Interlocking: stall pipeline for one or more cycles
  - Forwarding: two types of forwarding:
    - The ALU result of Instr$_1$ in EX stage can immediately be forwarded back to ALU input of EX stage as an operand for Instr$_2$.
    - The memory load data from MEM stage can be forwarded to ALU input of EX stage.
  - Forwarding with interlocking:
    - Assuming that Instr$_2$ is data dependent on the load instruction Instr$_1$ then Instr$_2$ has to be stalled until the data loaded by Instr$_1$ becomes available.
Hardware solution to data dependency hazard – interlocking

IF | ID | EX | MEM | WB
---|----|----|-----|------
add Reg2,Reg1,Reg2

IF | ID | EX | MEM | WB
mul Reg1,Reg2,Reg1

Register Reg2

bubbles

time
Solution to data dependency hazard - forwarding
Example of data dependency hazard not resolvable by forwarding

load Reg2, B

add Reg2, Reg1, Reg2

not possible!
Example of data dependency hazard – resolved by forwarding and stalling
Control hazards occur when a wrong fetch decision results in a new instruction fetch and the pipeline being flushed.

Dealing with control hazards may require significant hardware overheads.

Solutions include:

- Multiple Pipeline streams
- Prefetching the branch target
- Using a Loop Buffer
- Branch Prediction
- Delayed Branch
- Reordering of Instructions
- Multiple Copies of Registers (backups)
3- stage ARM Cortex M3 pipeline

- Each instruction is executed in three stages:
  - Fetch – instruction is fetched from memory and placed in pipeline;
  - Decode – instruction is decoded and data-path signals prepared for next cycle;
  - Execute – instruction reads ALU operands from registers and writes result to destination register.

- ARM pipeline is linear: processor throughput is one instruction per clock cycle while individual instruction takes three clock cycles.
  - When a branch instruction is fetched, a pipeline faces difficulties, because wrong instructions may have been fetched into the pipeline. If this occurs, pipeline flushes and has to be refilled. ARM pipeline maintains high performance and predicts branch behaviour because PC address is calculated 2 instructions ahead of current instruction.
MIPS Pipeline

• Pipelined version of MIPS is a good example of a pipelined RISC architecture (relatively simple and real)

• We will first recall the single-cycle (non-pipelined) MIPS architecture and its data path organisation

• And we will look how MIPS pipeline works
### MIPS (RISC) Instructions

<table>
<thead>
<tr>
<th>OP</th>
<th>Description</th>
<th>OP</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>Load Byte</td>
<td>MULT</td>
<td>Multiply</td>
</tr>
<tr>
<td>LBU</td>
<td>Load Byte Unsigned</td>
<td>MULTU</td>
<td>Multiply Unsigned</td>
</tr>
<tr>
<td>LH</td>
<td>Load Halfword</td>
<td>DIV</td>
<td>Divide</td>
</tr>
<tr>
<td>LHU</td>
<td>Load Halfword Unsigned</td>
<td>DIVU</td>
<td>Divide Unsigned</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
<td>MFHI</td>
<td>Move From HI</td>
</tr>
<tr>
<td>LWL</td>
<td>Load Word Left</td>
<td>MTHI</td>
<td>Move To HI</td>
</tr>
<tr>
<td>LWR</td>
<td>Load Word Right</td>
<td>MFLQ</td>
<td>Move From LO</td>
</tr>
<tr>
<td>SB</td>
<td>Store Byte</td>
<td>MTLO</td>
<td>Move To LO</td>
</tr>
<tr>
<td>SH</td>
<td>Store Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>Store Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWL</td>
<td>Store Word Left</td>
<td>J</td>
<td>Jump</td>
</tr>
<tr>
<td>SWR</td>
<td>Store Word Right</td>
<td>JAL</td>
<td>Jump and Link</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JR</td>
<td>Jump to Register</td>
</tr>
<tr>
<td></td>
<td>Arithmetic Instructions (ALU Immediate)</td>
<td>JALR</td>
<td>Jump and Link Register</td>
</tr>
<tr>
<td>ADDI</td>
<td>Add Immediate</td>
<td>BEQ</td>
<td>Branch on Equal</td>
</tr>
<tr>
<td>ADDIU</td>
<td>Add Immediate Unsigned</td>
<td>BNE</td>
<td>Branch on Not Equal</td>
</tr>
<tr>
<td>SLTI</td>
<td>Set on Less Than Immediate</td>
<td>BLEZ</td>
<td>Branch on Less Than or Equal to Zero</td>
</tr>
<tr>
<td>SLTIU</td>
<td>Set on Less Than Immediate Unsigned</td>
<td>BGTZ</td>
<td>Branch on Greater Than Zero</td>
</tr>
<tr>
<td>ANDI</td>
<td>AND Immediate</td>
<td>BLT</td>
<td>Branch on Less Than</td>
</tr>
<tr>
<td>ORI</td>
<td>OR Immediate</td>
<td>BGEZ</td>
<td>Branch on Greater Than or Equal to Zero</td>
</tr>
<tr>
<td>XORI</td>
<td>Exclusive-OR Immediate</td>
<td>BLTZAL</td>
<td>Branch on Less Than Zero And Link</td>
</tr>
<tr>
<td>LUI</td>
<td>Load Upper Immediate</td>
<td>BGEZAL</td>
<td>Branch on Greater Than or Equal to Zero And Link</td>
</tr>
<tr>
<td></td>
<td>Arithmetic Instructions (3-operand, R-type)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>LWCz</td>
<td>Load Word to Coprocessor</td>
</tr>
<tr>
<td>ADDU</td>
<td>Add Unsigned</td>
<td>SWCz</td>
<td>Store Word to Coprocessor</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>MTCz</td>
<td>Move To Coprocessor</td>
</tr>
<tr>
<td>SUBU</td>
<td>Subtract Unsigned</td>
<td>MFCz</td>
<td>Move From Coprocessor</td>
</tr>
<tr>
<td>SLT</td>
<td>Set on Less Than</td>
<td>CTCz</td>
<td>Move Control To Coprocessor</td>
</tr>
<tr>
<td>SLTU</td>
<td>Set on Less Than Unsigned</td>
<td>CFCz</td>
<td>Move Control From Coprocessor</td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
<td>COPz</td>
<td>Coprocessor Operation</td>
</tr>
<tr>
<td>OR</td>
<td>OR</td>
<td>BCzT</td>
<td>Branch on Coprocessor z True</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive-OR</td>
<td>BCzF</td>
<td>Branch on Coprocessor z False</td>
</tr>
<tr>
<td>NOR</td>
<td>NOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLL</td>
<td>Shift Left Logical</td>
<td>SYSCALL</td>
<td>System Call</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Right Logical</td>
<td>BREAK</td>
<td>Break</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Right Arithmetic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLLV</td>
<td>Shift Left Logical Variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRLV</td>
<td>Shift Right Logical Variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRAV</td>
<td>Shift Right Arithmetic Variable</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Special Instructions**

- B (I type)
- T (I type)
- J (J type)
- K (J type)

**Multiply/Divide Instructions**

- (FP type)

**Jump and Branch Instructions**

- B (I type)

**Coprocessor Instructions**

- LWCz
- SWCz
- MTCz
- MFCz
- CTCz
- CFCz
- COPz
- BCzT
- BCzF
MIPS Instruction Formats

I-type (immediate)

6 5 5 16
Operation rs rt Immediate

J-type (jump)

6 26
Operation Target

R-type (register)

6 5 5 5 5 6
Operation rs rt rd Shift Function

Operation  Operation code
rs  Source register specifier
rt  Source/destination register specifier
Immediate  Immediate, branch, or address displacement
Target  Jump target address
rd  Destination register specifier
Shift  Shift amount
Function  ALU/shift function specifier

FLOATING POINT INSTRUCTION FORMATS

<table>
<thead>
<tr>
<th>Format</th>
<th>opcode</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR</td>
<td>31 26 25</td>
<td>21 20</td>
<td>16 15</td>
<td>11 10</td>
<td>6 5</td>
<td>0</td>
</tr>
<tr>
<td>FI</td>
<td>31 26 25</td>
<td>21 20</td>
<td>16 15</td>
<td>immediate</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
NB: This diagram does not show J (absolute jump) instruction data path
Single-cycle MIPS with control signals shown
Single-cycle MIPS with Instruction Decoder (Control) shown
Single-cycle MIPS with J data path shown
MIPS Pipeline Stages

IF: Instruction fetch
ID: Instruction decode/register file read
EX: Execute/address calculation
MEM: Memory access
WB: Write back
MIPS Pipeline registers and data paths

IF | ID | EX | MEM | WB

Diagram showing the pipelined stages of instruction processing in a MIPS processor.
Pipelined MIPS with control signals shown