Lesson 1 Course Notes

Review of Computer Architecture

Embedded Systems ideal: low power, low cost, high performance

Overview of VLIW and ILP

What is ILP?
It can be seen in:
  - Superscalar
  - In Order Processors
  - VLIW
  - In and Out of Order Processors

For Embedded Systems VLIW, in order processors are good because multiple instructions in a slot lead to better performance with the most energy efficient solution.

Review of ILP:
ILP is the programs’ property - some programs have a high ILP, some a lower one. ILP is independent of the hardware. ILP means executing multiple operations simultaneously.

In embedded systems there is a trade-off between performance and power consumption. Using ILP saves power and leads to DECREASING clock frequency.

\[
\frac{1}{e} \rightarrow f^2
\]

\[
\left(\frac{1}{2f}\right)^2 \rightarrow \frac{1}{4f^2} \rightarrow 4e \ldots \text{Reducing the frequency by half leads to 4 times the energy.}
\]

Sequential Program Semantics
For a 5 stage pipeline, we should expect to be able to execute an instruction each clock cycle. But many instructions take more than one cycle to complete. As a result, there are bubbles in the pipeline as the processor waits for instructions to complete.

For a single thread:
- the processor tries to issue an instruction every clock cycle
- but dependencies, control hazards, and long latency instructions will reduce the number of instructions executed.
- delay result in execution of < 1 instruction per cycle on average
To mitigate this, ILP is used.

IPC = Instructions Per Cycle

Low IPC is result of:
- strict sequential semantics
- instructions stalled for data or resource release by other instructions

**Simple VEX Example**

NOPs are added to create delays in the pipeline so that instructions can complete

```plaintext
ldw $r20 = 0[$r20]
nop
nop
cmpeq $b0 = $r20, 0
nop
br $b0, DONE
ldw $r23 = 0[$r13]
nop
sub $r23 = $r23, $r20
add $r4 = $r4, $r25
```

In this example, there are NOPs after the load and compare instructions because these instructions take 2 and 1 cycles respectively.

The program takes a total of 11 clock cycles. There are 6 instructions....

IPC = (clock cycles)/(number of instructions) = 11/6 = 5.5

We use pipelining to overlap instructions.

The non-pipelined architecture takes way more clock cycles than the pipelined architecture.

Pipelines in modern CPUs have the following characteristics:
- execution stages are divided into several steps
- a later operation can share the resources used by the first operation in previous cycles
- shared hardware can be pipelined
The result: instructions can be overlapped

**IPC Quiz**
In a typical multi-cycle MIPS machine:
A load takes 5 cycles
A store takes 4 cycles
An R-type instruction takes 4 cycles
Branch takes 3 cycles
Jump takes 3 cycles

If a program has the following characteristics, what is the CPI (cycles per instruction)?

50% R-type, 10% load, 20% store, 8% branch, 29% jump instructions

R-type instructions: 4 cycles, 50% = 4*50
load instructions: 5 cycles, 10% = 5 * 10
store instructions: 4 cycles, 20% = 4 * 20
branch instructions: 3 cycles, 8% = 3 * 8
jump instructions: 3 cycles, 2% = 3 * 2

CPI = (4*50 + 5*10 + 4*20 + 3*8 + 3*2)/100 = 3.6

CPI is specific to a program. The mix of instructions determines the CPI.

**Basic Pipeline Execution Model**
Fetch - Decode - Execute - Memory Access - Write Back

**Pipeline Execution Model Quiz**

If we were to observe the end of the pipeline, for how many cycles initially will we see not 0 single instruction coming out? Assume the pipeline is k stages deep.

It takes k-1 cycles to fill the pipeline.
Example Pipeline Revised by VEX
The VEX processor can have up to 4 operations in each cycle using VLIW.

For the instructions in a single word, the instructions:
- cannot have dependencies
- cannot share resources
- only one memory access per word

Independent loads are hoisted above branches
Dismissable loads are supported (dw.d = dismissable loads, like a conditional move)
This means the loads can start before the branch outcome is known.

Since loads have latencies, at least two nops are included to allow for the delay.

```
ldw $r20 = 0[$r20]
nop
nop
cmpeq $b0 = $r20, 0
nop
br $b0, DONE
ldw $r23 = 0[$r13]
nop
nop
sub $r23 = $r23, $r20
add $r4 = $r4, $r23
```

In VLIW this can be changed.

The loads in slot 1 is a regular load, but the loads in slots 2 and 3 are dismissable loads.
The fourth slot cannot be used to hold the cmpeq instruction because it needs the results of the first load - which takes 2 clock cycles to become available.

The VLIW pipeline can be filled in the following manner.

<table>
<thead>
<tr>
<th></th>
<th>ldw $r20 = 0($r10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ldw.d $r21 = 0($r11)</td>
</tr>
<tr>
<td>3</td>
<td>ldw.d $r22 = 0($r12)</td>
</tr>
<tr>
<td>4</td>
<td>ldw.d $r23 = 0($r13) cmpeq $b0 = $r20, 0</td>
</tr>
<tr>
<td>5</td>
<td>ldw.d $r24 = 0($r14) cmpeq $b1 = $r21, 0</td>
</tr>
<tr>
<td>6</td>
<td>ldw.d $r25 = 0($r15) cmpeq $r2 = $r22, 0 br $b0, DONE</td>
</tr>
<tr>
<td>7</td>
<td>sub $r23 = $r23, $r20</td>
</tr>
<tr>
<td>8</td>
<td>add $r4 = $r4, $r23 sub $r24 = $r24, $r21 br $b1, DONE</td>
</tr>
<tr>
<td>9</td>
<td>add $r4 = $r4, $r24 sub $r25 = $r25, $r22 br $b2, DONE</td>
</tr>
<tr>
<td>10</td>
<td>add $r4 = $r4, $r25</td>
</tr>
</tbody>
</table>

Loads cannot go in the same VLIW word.
The loads (ld.d) are dismissable because they are dependent on a branch condition that is not known. Therefore the load cannot be committed.

It takes 2 cycles to get data from a load, so all instructions dependent upon loads must wait two cycles.
The IPC for this program is:
18 instructions, 10 cycles = 1.8 with VLIW
18 instructions, 33 cycles = .55 with sequential program

**Branches**

Branches cause degradation of IPC. Through bubbles and flushes.

Using branch predictors is expensive for embedded systems.
Using VLIW, branches can be moved up and/or bubbles can be filled with instructions that are independent of the branch. These methods are limited if the branch cannot be moved very much and if there are not enough independent instructions to fill the empty slots.

**Minimizing Dependences**

Minimizing dependences can be done to increase the number of independent instructions.

Types of dependences:
Control: outcome of branches
Data:
- RAW - true dependence
- WAW - false dependence
- WAR - false dependence

![Diagram](image)
Minimizing Dependences Techniques

- Branch prediction: predict which branch might be taken
- Predication: use conditional moves
- Register Renaming: used to avoid WAW and WAR dependences

```
if (p) then
  S1
else
  S2;
endif;
S3
```

The instructions in S3 are not dependent on p. S3 instructions will be executed regardless of the outcome of p.

Other Parallelisms

- Vector processing: good for regular code containing long vectors
  Not good for irregular codes. Not used for embedded systems

- Multithreading and Multiprocessing: hyper threading is used. Low in terms of energy consumption but high in hardware overhead

- Micro-SMD: single instruction, multiple data. Each register is a collection of smaller data. It is energy efficient, and is slowly being adopted in embedded systems.