CPU Structure and Function

Ch 12 [Sta10]
- Registers
- Instruction cycle
- Pipeline
- Dependences
- Dealing with Branches

Registers
- Top of memory hierarchy
- User visible registers
  - Programmer / Compiler decides how to use these
  - How many? Names?
- Control and status registers
  - Some of these used indirectly by the program
    - PC, PSW, flags, ...
  - Some used only by CPU internally
    - MAR, MBR, ...
- Internal latches (apurekisteri) for temporal storage during instruction execution
  - Example: Instruction register (IR) instruction interpretation; operand first to latch and only then to ALU
  - ALU output before result moved to some register

User visible registers
- Different processor families
  - Different number of registers
  - Different naming conventions
  - Different purposes
- General-purpose registers (yleisrekisterit)
- Data registers (datarekisterit)
- Address registers (osoiterekisterit)
- Segment registers (segmentirekisterit)
- Index registers (indeksirekisterit)
- Stack pointer (pino-osoitin)
- Frame pointer (ympäristöosoitin)
- Condition code registers (tilarekisterit)

Example

PSW - Program Status Word
- Name varies in different architectures
- State of the CPU
  - Privileged mode vs user mode
  - Result of comparison (vertailu)
    - Greater, Equal, Less, Zero, ...
  - Exceptions (poikkeus) during execution?
    - Divide-by-zero, overflow
    - Page fault, “memory violation”
- Interrupt enable/disable
  - Each ‘class’ has its own bit
- Bit for interrupt request?
  - I/O device requesting guidance

General structure of CPU
Instruction cycle (käskysyöksi)

Instruction fetch (käskyn nouto)

Operand fetch, Indirect addressing (Operandin nouto, epäsuora osoltus)

Data flow, interrupt cycle

Computer Organization II

Instruction pipelining (liukuhihna)

Laundry example (by David A. Patterson)
Sequential Laundry

- Takes 6 hours for 4 loads:

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 PM</td>
<td>30</td>
<td>40</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>20</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>40</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>20</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

  - If they learned pipelining, how long would laundry take?

  - Average latency (calculate,Keith,viive):
    - 1.5 h per load
  - Throughput (Läpimenoaste):
    - 0.67 loads per h

Pipelined Laundry

- Takes 3.5 hours for 4 loads

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 PM</td>
<td>30</td>
<td>40</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>20</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>40</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>20</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

  - At best case, one load is completed every 40 minutes! (0.67 h / finished load)

Lessons

- Pipelining does not help latency of single task, but it helps throughput of the entire workload
- Pipelining can delay single task compared with situation where it is alone in the system
- Next stage occupied, must wait
- Multiple tasks operating simultaneously, but different phases
- Pipeline rate limited by slowest pipeline stage
  - Can proceed when all stages done
  - Not very efficient, if different stages have different durations, unbalanced lengths
- Potential speedup
  - Maximum possible speedup
  - Number of pipe stages

Lessons

- Complex implementation,
- May need more resources
  - Enough electrical current and sockets to use both washer and dryer simultaneously
  - Two (or three) people present all the time in the laundry
  - 3 laundry baskets
  - Time to "fill" pipeline and time to "drain" it reduce speedup
  - Resources are not fully utilized
  - "Hiccups" (hikka)
  - Variation in task arrivals, works best with constant flow of tasks

2-stage instruction execution pipeline (2-vaiheinen liukuhihna)

- Instruction prefetch (ennaltanouto) at the same time as execution of previous instruction
- Principle of locality: assume ‘sequential’ execution
- Problems
  - Execution phase longer = fetch stage sometimes idle
  - Execution modifies PC (jump, branch) = fetched wrong instr.
    - Prediction of the next instruction’s location was incorrect!
- Not enough parallelism = more stages?

Discussion

6-Stage (6-Phase) Pipeline

- Instruction prefetch (ennaltanouto)
- Instruction decode (parantalaisuus)
- Execute instruction (hoitopyynti)
- Write operands (läpimenoaste)

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>E8</td>
<td>E8</td>
<td>CO</td>
<td>FO</td>
<td>FO</td>
<td>EI</td>
<td>WQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 2</td>
<td>FE</td>
<td>DB</td>
<td>CO</td>
<td>FO</td>
<td>FO</td>
<td>EI</td>
<td>WQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 3</td>
<td>FE</td>
<td>DB</td>
<td>CO</td>
<td>FO</td>
<td>FO</td>
<td>EI</td>
<td>WQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 4</td>
<td>FE</td>
<td>DB</td>
<td>CO</td>
<td>FO</td>
<td>FO</td>
<td>EI</td>
<td>WQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 5</td>
<td>FE</td>
<td>DB</td>
<td>CO</td>
<td>FO</td>
<td>FO</td>
<td>EI</td>
<td>WQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 6</td>
<td>FE</td>
<td>DB</td>
<td>CO</td>
<td>FO</td>
<td>FO</td>
<td>EI</td>
<td>WQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 7</td>
<td>FE</td>
<td>DB</td>
<td>CO</td>
<td>FO</td>
<td>FO</td>
<td>EI</td>
<td>WQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 8</td>
<td>FE</td>
<td>DB</td>
<td>CO</td>
<td>FO</td>
<td>FO</td>
<td>EI</td>
<td>WQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 9</td>
<td>FE</td>
<td>DB</td>
<td>CO</td>
<td>FO</td>
<td>FO</td>
<td>EI</td>
<td>WQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Discussion

6.2.2012
Lecture 7: CPU structure and function

Pipeline speedup (nopeutus)?

- Let’s calculate (based on Fig 12.10):
  - 6-stage pipeline, 9 instr. → 14 time units total
  - Same without pipeline → 96 time units total
  - Speedup = Time_{pipeline} / Time_{no pipeline} = 54/14 = 3.86 < 6!
  - Maximum speed at times 6-14
    - One instruction per time unit finishes
    - 8 time units → 8 instruction completions
    - Maximum speedup = Time_{pipeline} / Time_{no pipeline} = 48/6 = 8

- Not every instruction uses every stage
  - Will not affect the pipeline speed – some stages unused
  - Speedup may be small (some stages idle, waiting for slow)
  - Serial execution could be faster (no wait for other stages)

Pipeline performance: one cycle time

\[ \tau = \max_i \left[ \frac{\tau_i}{k} \right] + d = \tau_m + d \implies d \]

- Cycle time is the same for all stages
- Each stage (phase) takes one cycle time to execute
- Slowest stage determines the pace

Maximum speed at times 6-14

- One instruction per time unit finishes
- 8 time units
- 8 instruction completions
- Maximum speedup = Time_{pipeline} / Time_{no pipeline} = 48/8 = 6

Pipeline Speedup

\[ \text{n instructions, k stages, } t = \text{cycle time} \]

No pipeline:

\[ T_1 = nk \tau \]

Pipeline:

\[ T_k = k + (n-1) \tau \]

\text{k-stages before the first task (instruction) is finished}

Speedup:

\[ S_k = \frac{T_1}{T_k} = \frac{nk \tau}{k + (n-1) \tau} = \frac{nk}{k + (n-1)} \]

See Sta10 Fig 12.10 and check yourself!

Speed vs. nr stages vs. instructions w/no jumps?

More gains from multiple stages when more instructions without jumps

Pipeline Features

- Extra issues
  - CPU must store ‘midresults’ somewhere between stages and move data from buffer to buffer
  - From one instruction’s viewpoint the pipeline takes longer time than single execution
- But still
  - Executing large set of instructions is faster
  - Better throughput (instructions/sec)

The parallel (concurrent) execution of instructions in the pipeline makes them proceed faster as whole, but slows down execution of single instruction

Pipeline Problems and Design Issues

- Structural dependency (rakenteellinen riippuvuus)
  - Several stages may need the same HW
  - Memory used by FI, FO, WO?
  - ALU used by CO, EI?
- Control dependency (kontrollin riippuvuus)
  - No knowledge on next instruction
  - E.g., (conditional) branch destination may be known only after EI-stage
    - Prefetched and executed wrong instructions?
- Data dependency (datarriippuus)
  - E.g., instruction needs the result of the previous non-finished instruction

Comp. Org II, Spring 2012
Pipeline Dependency Problem Solutions

- In advance: prevent (some) dependency problems completely
  - Structural dependency
    - More hardware, e.g., separate ALUs for CO and EI stages
    - Lots of registers, less operands from memory
  - Control dependency
    - Clear pipeline, fetch new instructions
    - Branch prediction, prefetch and execute these, those, or both?
  - Data dependency
    - Change execution order of instructions
    - By-pass (oikopolku) in hardware between stages: earlier instruction’s result can be accessed already before its WO-stage is done
- At run time: Hardware must notice and wait until all possible dependencies are cleared
  - Add extra ‘waits’, ‘bubbles’, to the pipeline: Commonly used
  - Bubble (kupla) delays everything behind it in all stages

Data dependency

- Read after Write (RAW) (a.k.a true or flow dependency)
  - Occurs if succeeding read takes place before the preceding write operation is complete
- Write after Read (WAR) (a.k.a antidependency)
  - Occurs if the succeeding write operation completes before the preceding read operation takes place
- Write after Write (WAW) (a.k.a output dependency)
  - Occurs when the two write operations take place in the reversed order of the intended sequence

The WAR and WAW are possible only in architectures where the instructions can finish in different order.

Example: Data Dependency - RAW

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Result</th>
<th>Next</th>
<th>Depend.</th>
<th>Depend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL R1, R2, R3</td>
<td>1</td>
<td>FT</td>
<td>DCO</td>
<td>FE</td>
<td>WO</td>
</tr>
<tr>
<td>ADD R4, R5, R6</td>
<td>2</td>
<td>FT</td>
<td>DCO</td>
<td>FE</td>
<td>EI</td>
</tr>
<tr>
<td>SUB R7, R1, R8</td>
<td>3</td>
<td>FT</td>
<td>DCO</td>
<td>FE</td>
<td>EI</td>
</tr>
<tr>
<td>ADD R1, R1, R3</td>
<td>11</td>
<td>FT</td>
<td>DCO</td>
<td>FE</td>
<td>EI</td>
</tr>
</tbody>
</table>

Example: Change instruction execution order

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Result</th>
<th>Next</th>
<th>Depend.</th>
<th>Depend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL R1, R2, R3</td>
<td>1</td>
<td>FT</td>
<td>DCO</td>
<td>FE</td>
<td>WO</td>
</tr>
<tr>
<td>ADD R4, R5, R6</td>
<td>2</td>
<td>FT</td>
<td>DCO</td>
<td>FE</td>
<td>EI</td>
</tr>
<tr>
<td>SUB R7, R1, R8</td>
<td>3</td>
<td>FT</td>
<td>DCO</td>
<td>FE</td>
<td>EI</td>
</tr>
<tr>
<td>ADD R9, R0, R8</td>
<td>10</td>
<td>FT</td>
<td>DCO</td>
<td>FE</td>
<td>EI</td>
</tr>
</tbody>
</table>

Example: By-pass (oikopolku)

- New wires (and temp registers, latches) in pipeline
  - E.g., instr. result available to FO phase directly from phase EI

Pipelining and Jump Optimization

- Multiple streams (Monta suorituspolkuu)
- Delayed branch (Viivästetty hyppy)
- Prefetch branch target (Kohteen ennaltamonto)
- Loop buffer (Silmukkapuskin)
- Branch prediction (Erinnoituslogikka)
**Effect of Conditional Branch on Pipeline**

- Many phases, many useful instructions after branch instructions (to delay slots)
- Instructions in delay slots are fully executed!
- No roll-back of instructions needed due to incorrect prediction
- If no useful instruction available, compiler uses NOP
- Less actual work lost if branch occurs
- Next instruction almost done, when branch decision known
- This is easier than emptying the pipeline during branch
- Worst case: NOP-instructions waste some cycles

- Compiler places some useful instructions (1 or more) after branch instructions (to delay slots)
- No roll-back of instructions needed due incorrect prediction
- If no useful instruction available, compiler uses NOP
- Less actual work lost if branch occurs
- Next instruction almost done, when branch decision known
- This is easier than emptying the pipeline during branch
- Worst case: NOP-instructions waste some cycles

**Multiple instruction streams**

- Execute speculatively to both directions
- Prefetch instructions that follow the branch to the pipeline
- Prefetch instructions from branch target to (another) pipeline
- After branch decision: reject the incorrect pipeline (its results, changes)

**Problems**

- Branch target address known only after some calculations
- Second split on one of the pipelines
  - Continue any way? Only one speculation at a time?
- More hardware!
  - More pipelines, speculative results (registers), control
  - Speculative instructions may delay real work
  - Bus and register contention? More ALUs?
- Capability to cancel not-taken instruction stream from pipeline
  - Easier, if all changes done in WB phase

**Loop buffer**

- Keep $n$ most recently fetched instructions in high speed buffer inside the CPU
- Use prefetch also
  - With good luck the branch target is in the buffer
  - F. ex. IF-THEN and IF-THEN-ELSE structures
- Works for small loops (at most $n$ instructions)
- Fetch from memory just once
- Gives better spacial locality than just cache

**Prefetch branch target**

- Prefetch just branch target instruction, but do not execute it yet
- Do only FI-stage
- If branch taken, no need to wait for memory
- Must be able to clear the pipeline
- Prefetching branch target may cause page-fault

**Delayed Branch**

- Compiler places some useful instructions (1 or more) after branch instructions (to delay slots)
- Instructions in delay slots are always fully executed!
- No roll-back of instructions needed due incorrect prediction
- If no useful instruction available, compiler uses NOP
- Less actual work lost if branch occurs
- Next instruction almost done, when branch decision known
- This is easier than emptying the pipeline during branch
- Worst case: NOP-instructions waste some cycles
- Can be difficult to do (for the compiler)

**Static Branch Prediction**

- Make an (educated?) guess on which direction is more probable:
  - Branch or no?
- Static prediction (staattinen ennustus)
  - Fixed: Always taken (aina hypätään)
  - Fixed: Never taken (ei koskaan hypätä)
    - \~ 50% correct
  - Predict by opcode (operaatiokoodin perusteella)
    - In advance decided which codes are more likely to branch
    - Compilers know this, and use it for better performance
    - For example, BLE instruction is commonly used at the end of stepping loop, guess a branch
    - \~ 75% correct [LILJ88]
Dynamic Branch Prediction

- Dynamic prediction
  - Make a guess based on earlier history for (this) branch
  - Logic: What has happened in the recent history with this instruction
    - Improves the accuracy of the prediction
  - Implementation: extra internal memory = branch history table
    - Instruction address (for this branch)
    - Branch target (instruction or address) – need this for quick action
    - Decision: taken / not taken

- Simple prediction based on just the previous execution
  - 1 bit memory is enough
  - Loops will always have one or two incorrect predictions

2-Bit Branch Prediction Logic for One Instruction

- Improved simple model
  - Don’t change the prediction with one misprediction
  - Based on two previous executions of this instruction
  - 2 bits enough

Branch Prediction History Table

- State and prediction: taken/not taken
- Branch address
- Branch target
- Label: prediction engine, instruction address
- Where to jump, if branch taken
- Address in instruction

Summary

- Pipeline basics
  - Stage length, pipeline fill-up and drain times
  - Response time, throughput, speedup

- Hazards, dependencies
  - Structural, control, data (RAW, WAR, WAW)
  - How to avoid before time?
  - How to handle at run time?

- How to minimize branch costs?
  - Delayed branch, multiple pipeline streams, prefetch branch target, loop buffer, branch prediction

Review Questions

- What information PSW needs to contain?
- Why 2-stage pipeline is not very beneficial?
- What elements effect the pipeline?
- What mechanisms can be used to handle branching?
- How does CPU move to interrupt handling?