TDTS08: whattolearn

Name	PIN	Email
Pål Kastman	851212-7575	palka285@student.liu.se
Alexander Yngve	930320-6651	aleyn573@student.liu.se

Contents

1 Introduction

This documents only purpose is to document what I will need to learn in the course tdts08.
YNGVE OM DU VILL BYTA RUBRIKERNA SÅ ÄR DET HELT OKEJ, DOM ÄR BARA DÄR SOM STÖD NU

2 Lecture 1: Memory Systems

Memory hierarchy
registers, cache, main memory etc.

2.1 Cache memory

how does it work? why do we use it? positive and negative things about it!

2.1.1 Locality of reference

Temporal- and spatial locality, differences and how we use these in caches.

2.1.2 Replacement methods

yes, there's different kinds of them.

2.1.3 Cache Design

One, two, three level caches, how does it work. up- and downsides of them.

2.1.4 Split Caches

2.1.5 Unified Caches

2.1.6 Direct mapping cache

2.1.7 Associative mapping cache

2.1.8 Fully Associative Organization

2.1.9 Write Policy

Write through
Write through with buffered write

Write back

2.2 Virtual memory

2.2.1 Paging

2.2.2 Page fault

2.2.3 Page replacement

2.2.4 Replacement algorithms

3 Lecture 3: Instruction pipelining

how does it work, and why do we need it?

3.1 Pipeline hazards

what problems do we have with pipelining

3.1.1 Structural (resource) hazards

what are they and how do we solve it?

3.1.2 Data hazards

3.1.3 Control hazards

3.2 Branch handling

3.2.1 Pre-fetch branch target

3.2.2 Loop buffer

3.2.3 Delayed branch

3.3 Branch prediction

3.3.1 Static Branch Prediction

3.3.2 Dynamic Branch Prediction

3.3.3 Bimodal Prediction

4 Lecture 4: RISC Computers

What are they, how do they work, and why do we need them?
Semantic gap
Problems with RISC

4.1 Program execution features

what are programs doing most of the time?

4.2 RISC characteristics

Small number of simple instructions
Execution of one instruction per clock cycle
Complex operations are executed as a sequence of simple instructions
only LOAD and STORE instructions reference data in memory
Only a few simple addressing modes are used
Instructions are of fixed length and uniform format.
Large number of registers, this is because the reduced complexity of the processor leaves silicon space on the chip to implement them (opposite of CISC).

4.2.1 Register Windows

Large number of registers is usually very useful.
However, if contents of all registers must be saved at every procedure call, more registers mean longer delay.
A solution to this problem is to divide the register file into a set of fixed-size windows.
– Each window is assigned to a procedure.
– Windows for adjacent procedures are overlapped to allow parameter passing

4.2.2 Main advantages of RISC

Best support is given by optimizing most used and most time consuming architecture aspects.
– Frequently executed instructions.
– Simple memory reference.
– Procedure call/return.
– Pipeline design.
Consequently, we have:
– High performance for many applications;
– Less design complexity;
– Reduced power consumption:
– reducing design cost and time-to-market (newer technology)

4.2.3 Criticism of RISC

Operation might take several instructions to accomplish
more memory access might be needed
Execution speed may be reduced for certain applications.

It usually leads to longer programs, which needs larger memory space to store.
It makes it more difficult to program machine codes and assembly programs.

4.3 RISC vs. CISC

In embedded processors RISC is the better choice

5 Lecture 5: Superscalar Architecture (SSA)

Computer designed to improve computation on scalars instructions.
A scalar is a variable that can hold only one atomic value at a time, e.g., an integer or a real.

A scalar architecture processes one data item at a time – the computers we discussed up till now.
Examples of non-scalar variables:
– Arrays – Vector Processor
– Matrices – Graphics Processing Unit (GPU)
– Records

In a superscalar architecture (SSA), several scalar instructions can be initiated simulaneously and executed independently.

5.1 Instruction-level parallelism

Most operations are on scalar quantities, speed up these operations will lead to large performance improvement.

5.2 Superpipelining

Divide pipelining stages into different several sub-stages, and hence increase the number of instructions which are handled by the pipeline at the same time.
A picture would be nice here

– For example by diving each stage into two sub stages, we will be able (in the ideal situation) to perform each stage at twice the stage.
– No duplication of hardware is needed.
– Not all stages can be divided into (equal length) sub stages.
– Hazards more difficult to resolve.
– More complex hardware.
– Interrupt handling and testing will be more complicated.

5.3 Difference between Superpipelined and Superscalar Designs

The difference is that Superpipeline can perform several instructions in one clock cycle, whereas Superscalar performs several instructions in parallel

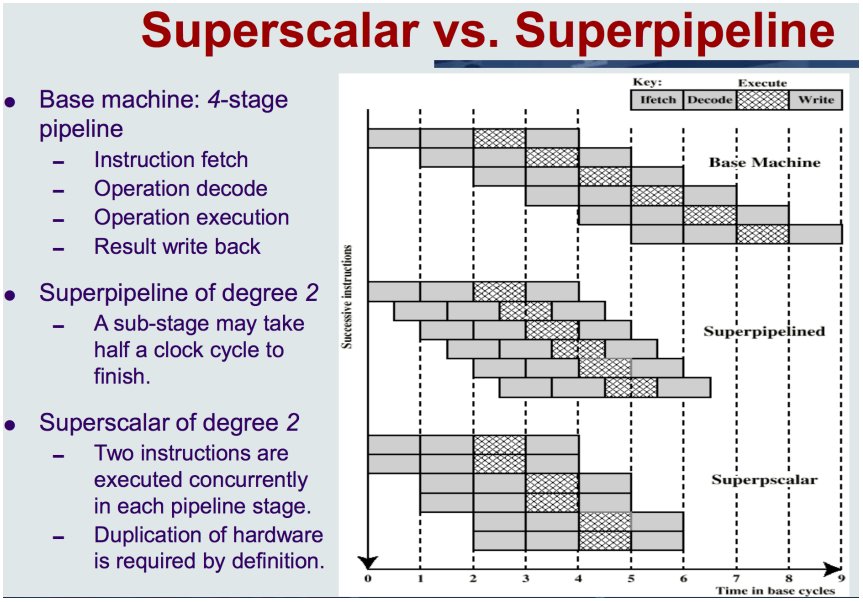

Figure 1 – Difference between a Superscalar and a Superpipeline architecture

5.4 Superscalar Superpipeline Design

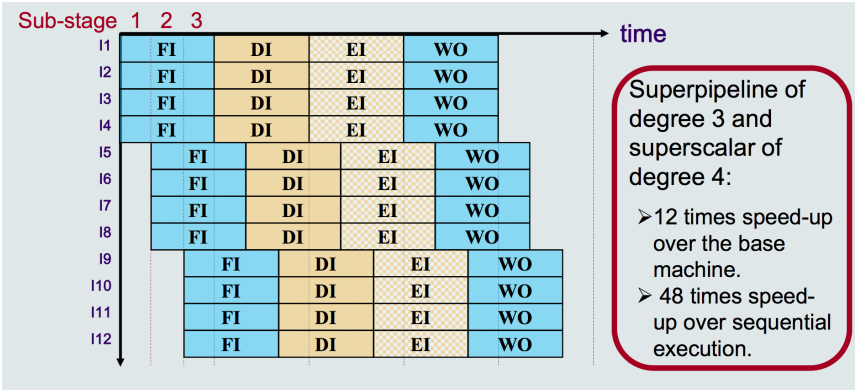

Figure 2 – Superscalar Superpipeline architecture

The new trend is to combined the two ideas. In Figure 2 it says that this would give us a 48 times speedup, this is not the case though because of data dependencies.

5.5 Dependency issues

The main problems are: – Resource conflicts.
– Control (procedural) dependency.

– Data dependencies.

These are very similar to the cases in normal pipelining (data hazards). But the consequences are more severe here because the parallelism are greater and thus larger amount of performance will be lost.

5.5.1 Resource Conflicts

Several instructions compete for the same hardware resource at the same time.
– For instance, two aritmethic instructions need the same floating-point unit for execution.
– similar to structural hazards in pipeline.

They can be solved partly by introducing several hardware units for the same functions.
– e.g. have two floating point units.
– the hardware units can also be pipelined to support several operations at the same time.
– however, memory units **can't be duplicated**.

5.5.2 Procedural Dependency

5.5.3 Data Conflicts

5.5.4 Window of execution

5.5.5 Data Dependency

5.5.6 True Data Dependency

5.5.7 Output Dependency

5.5.8 Anti Dependency

5.5.9 Effect of Dependencies

5.6 Parallel instruction execution

5.6.1 Instruction vs Machine Parallelism

5.6.2 Division and Decoupling

5.6.3 SSA instruction Execution Policies

5.6.4 In-Order with In-Order Completion

This seems important to understand

5.6.5 In-Order issue with Out-of-Order Completion

This seems important to understand

6 Lecture 6: VLIW Processors

The difference from Superscalar is that in that architecture the hardware solves everything for us, if improvements are made on the design we don't need to change the programs.
The down side though is that the architecture is very complex.
– A lot of hardware is needed for run-time detection of parallelism.
– It consumes a lot of power.
– There is, therefore a limit in how far we can go with this technique.

The instruction window for execution is limited in size.
– This limits the capacity to detect large number of parallel instructions.

6.1 Very Long Instruction Word Processors

Several operations that can be executed in parallel are placed in a single instruction word.
–VLIW rely on compile-time detecton of parallelism.
– The compiler analyzes the program and detects operations to be executed in parallel.

After one instruction has been fetched, all the corresponding operations are issued in paralell. The instruction window limitation disappears: the compiler can potentially analyze the whole program to detect parallel operations.

6.1.1 Explicit Parallelism

Instruction parallelism scheduled at compile time.
– Included within the machine instructions explicitly.

An EPIC (Explicitly Parallel Instruction Computing) processor
–uses this information to perform parallel execution.

The hardware is very much simplified
– The controller is similar to a simple scalar computer.
– The number of FUs can be increased without needing additional sophisticated hardware to detect parallelism, as in SSA.

Compiler has much more time to determine parallel operations.
– This analysis is only done once off-line, while run-time detection is carried out by SSA hardware for each execution of the code.
–Good compilers can detect parallelism based on global analysis of the whole program.

6.1.2 Main issues

– Need a large number of registers.
– Large data transport capacity is needed between Fus and the register files and between register files and memory.
– High bandwidth between instruction cache and fetch unit is also needed due to long iunstructions.

6.1.3 Software issues

6.2 Loop unrolling

6.3 IA-64 architecture

6.3.1 Predicated execution

6.3.2 Instruction format

6.3.3 Branch Predication

NOT THE SAME AS BRANCH PREDICTION

6.3.4 Placement of Loading

6.3.5 Speculative Loading

7 Parallel Processing

7.0.1 Why Parallel Processing?

7.0.2 Parallel Computer

7.0.3 Parallel Program

7.1 Architecture classification

7.1.1 Flynn's Classification of Architectures

– Single instruction, single data stream - **SISD**
– Single instruction, multiple data stream - **SIMD**
– Multiple instruction, single data stream - **MISD**
– Multiple instruction, multiple data stream- **MIMD**

GÅ IGENOM DESSA

7.2 Performance evaluation

7.3 Interconnection network

7.3.1 Single Bus

7.3.2 Completely Connected Network

7.3.3 Crossbar Network

7.3.4 Mesh Network

7.3.5 Hypercube Network

8 Lecture 8: SIMD Architectures

8.1 Vector Processors

– array processors
– vector processors
– data parallelism
Typical SISD processors who behave like SIMD processors

8.1.1 Instruction-level parallelism

8.1.2 Thread-level parallelism

8.1.3 Data parallelism

8.1.4 Vector Processors

8.2 Array Processors

Typical SIMD processors

8.3 Dedicated Memory Organization

8.4 Global Memory Organization

8.5 Cray supercomputers

8.5.1 Cray X1

8.6 Multimedia extensions

8.6.1 sub-word execution

8.6.2 Packed Data Types

8.6.3 SIMD Arithmetic Examples

9 Lecture 9: MIMD Architectures

A set of general purpose processors connected together
In contrast to a SIMD computer, a MIMD computer can execute different programs on different processors.

– Works asynchronously, and don't have to synchronize with each other.
– At any time, different processors may be executing different instructions on different pieces of data.
– They can be built from commodity (off-the-shelf) microprocessors with relatively little effort.
– They are also highly scalable, provided that an appropiate memory organization is used.
– Most current parallel computer are built based on the MIMD architecture.

9.0.1 SIMD vs. MIMD

9.0.2 MIMD Processor Classification

9.0.3 MIMD with Shared Memory

9.0.4 MIMD with Distributed Memory

9.0.5 Shared-Address-Space Platforms

9.0.6 Multi-Computer Systems

9.0.7 MIM Design Issues

9.1 Symmetric multiprocessors (SMP)

9.1.1 SMP Advantages

9.1.2 SMP based on Shared Bus

9.1.3 Multi-Port Memory SMP

9.1.4 Operation System Issues

9.1.5 IBM S/390

9.2 NUMA Architecture

9.2.1 Memory Access Approaches

9.3 Clusters

9.3.1 Losely Coupled MIMD - Clusters

9.3.2 Clusters benefits

9.3.3 Clusters Configurations

9.3.4 IBM Blue Gene Supercomputer

9.3.5 Parallelizing Computation

9.3.6 Google Applications

10 Lecture 10: Cache Coherence

To many section without text seems to bug LaTeX

10.1 Introduction

10.1.1 Write Through

10.1.2 Write Back

10.1.3 Software Solutions

10.1.4 Hardware Solutions

10.2 Directory protocols

10.2.1 Cache Coherence Operations

10.3 Snoopy protocols

10.3.1 Write Invalidate SP

10.3.2 Snoopy Cache Organization

10.3.3 MESI State Transition Diagram

10.3.4 Write Update SP

10.3.5 Invalidate vs. Update Protocols

10.3.6 Directory vs. Snoopy Schemes

10.4 L1-L2 consistence

10.4.1 Alpha-Server 4100

11 Lecture 11: Multi-Core and GPU

To many section without text seems to bug LaTeX

11.1 Multi-Core Computers

11.1.1 Intel Core i7

11.1.2 Superscalar vs. Multi-Core

11.1.3 Single Core vs. Multi-Core

11.1.4 Intel Polaris

11.2 Multithreading

11.2.1 Thread-Level Parallelism (TLP)

11.2.2 Scalar Processor Approaches

11.2.3 Superscalar Approaches

11.2.4 SMT and Chip Multiprocessing

11.2.5 Multithreading Paradigms

11.3 Graphic Processing Unit (GPU)

11.3.1 CPU vs. GPU

11.4 General Purpose GPUs

11.4.1 Divergent Execution

11.4.2 Reduce Branch Divergence

11.4.3 GPUPU

11.4.4 NVIDIA Tesla

11.4.5 CUDA Programming Language