

04 - Parallelism

Hans Dermot Doran: Institute of Embedded Systems

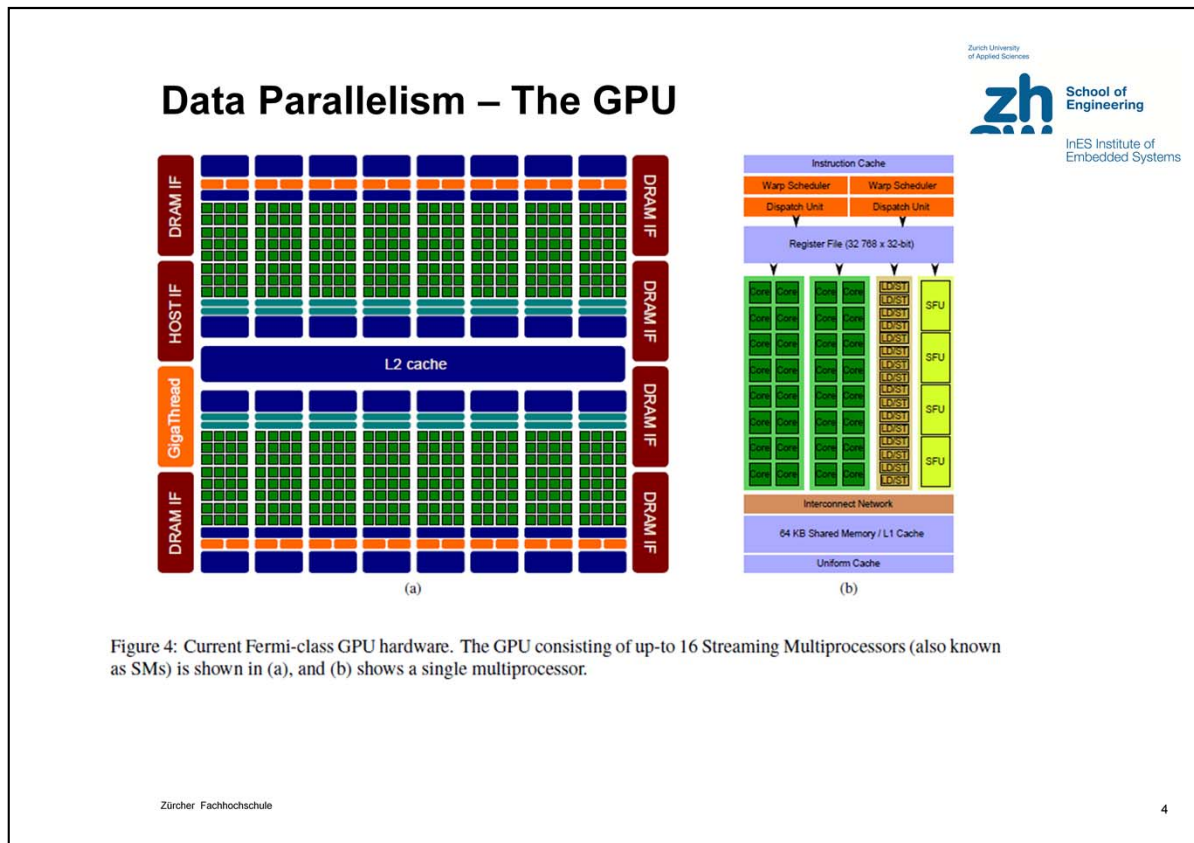
- V1.0 donn, Q4 2012
- V1.01 donn, 03.12.2012
- V2.0 donn, 12.05.2014 - Split into several files then co-joined with
parallelism
- V2.01 donn, 23.05.2014 - added text and reference, slide 10
- V2.02 donn, 12.05.2015 – added some slides
- V2.02 donn, 10.05.2016 – added some slides on pipelining and lockstep

What you should be able to do after today

- The student will be able to explain Data, Instruction and Task based parallelism and analyse CISC, RISC and DSP architectures according to these concepts
- The student will be able to schedule instructions on scalar and super-scalar architectures
- The student will be able to calculate long time effects on pipelines
- The student will be able to apply software pipelining
- The student will understand pipelining as applied to scheduling problems

Path to Glory

- Via Hardware Acceleration
 - Data Parallelism and GPU's
 - Task Parallelism: Co-Processors
 - Instruction Parallelism: Pipelines, Superscalar, DSP architectures
 - WCET and pipelines
 - Software Pipelining



GPU's are out of the scope of this module but for completeness sake: A GPU seeks to exploit data parallelism by executing the same instruction on multiple pieces of data in parallel. It does this in twofold fashion: 1.) Single Instruction Multiple Data. One instruction fetch – multiple results from independent pieces of data. Single Instruction Multiple Threads – split execution over multiple threads.

A simple introduction:

courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf

A GPU architecture is shown above – There is a big distance between this and simple HA but the principle is the same –

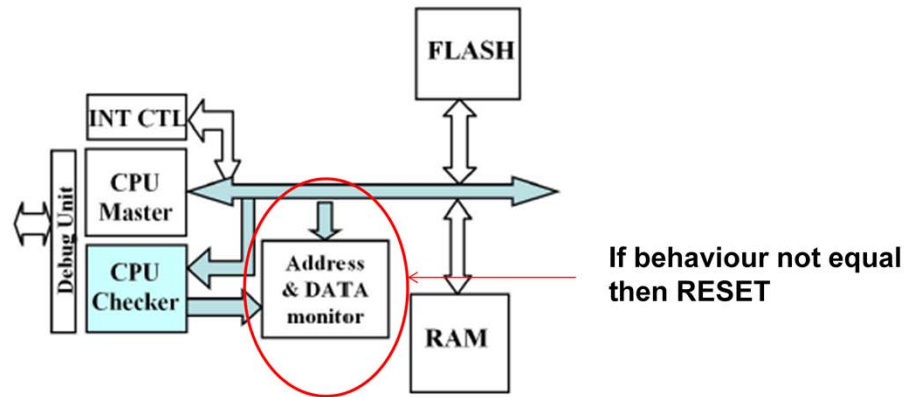
The pictures above From: André R. Brodtkorb, Trond R. Hagen, Martin L. Sætra, Graphics processing unit (GPU) programming strategies and trends in GPU computing

Teng Li, Vikram K. Narayana and Tarek El-Ghazawi, Exploring Graphics Processing Unit (GPU) Resource Sharing

Efficiency for High Performance Computing

Jonathan Palacios, Josh Triska, "A Comparison of Modern GPU and CPU Architectures: And the Common Convergence of Both", *Oregon State University*, Oregon State University, 2011. is a decent paper – the riposte (from Intel) may also be worth a read: Lee et. Al. "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU"

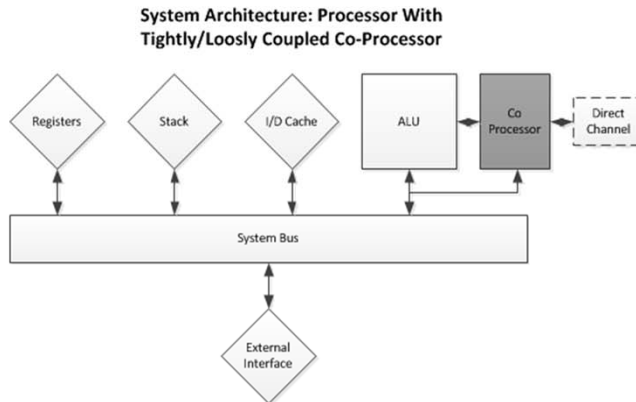
Lock-Step Processors



The idea of a lock-step processors is found in another context – specifically the realm of safe design and development. Lock-step processors are two tightly coupled processors – operating with in-phase clocks and carrying out the same instructions in parallel. One processor – the master – fetches the code and data and executes whereas the second processor executes the code and the monitor checks that the two processors do the same thing.

See: <http://www.acsel-lab.com/Publications/Papers/20-coproc-commssync-EURO88.pdf>

Instruction Parallelism (1) – Co-processors



A classical Co-Processor is a device that has its own instruction set. The use of this device is triggered by co-processor instructions in the program code. One mechanism is as follows:

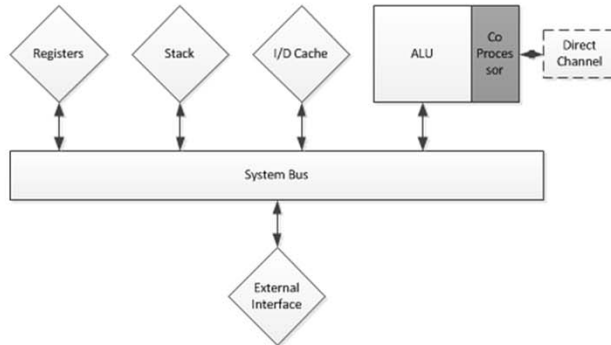
Loosely coupled co-processors have access to the same memory and external resources as the main processor through the system bus arbitration scheme. An older example is the 8087 numeric processor.

Loosely coupled is a dangerous word to use with the 8086/x87 family because some authors call this a tightly coupled processor. This is based on the idea that the processors operate in lock-step, that is they run with the same frequency. Both units read the code. If there is a FPU ESC code attached to the opcode then the “normal” processor ignores the opcode and the FPU takes over. Otherwise the FPU ignores the opcode code and the normal processor takes it. In this case as the co-processors are in lock-step if the FPU needs time to complete an instruction then the FXP (fixed-point processor) has to wait.

The 80387 took a different view – the processor which had the instruction fetch mechanism and appropriate queues – didn’t have to run at the same speed as the FPU but since it had the instruction fetch and decode unit the processor passed the instruction to the co-processor. If the instruction could execute without further hinderance it then did so.

Instruction Parallelism (2) – Co-processors

System Architecture: Processor With Tightly Coupled Co-Processor



Coprocessor instructions

This section contains the following subsections:

- *CDP and CDP2* on page 3-125
Coprocessor Data Operations.
- *MCR, MCR2, MCRR, and MCRR2* on page 3-126
Move to Coprocessor from ARM Register or Registers, possibly with coprocessor operations.
- *MRC, MRC2, MRRC and MRRC2* on page 3-127
Move to ARM Register or Registers from Coprocessor, possibly with coprocessor operations.
- *MSR* on page 3-128
Move to system coprocessor from ARM register.
- *MRS* on page 3-129
Move to ARM register from system coprocessor.
- *STS* on page 3-130
Execute system coprocessor instruction.
- *LDC, LDC2, STC, and STC2* on page 3-131
Transfer data between memory and Coprocessor.

Note
A coprocessor instruction causes an Undefined Instruction exception if the specified coprocessor is not present, or if it is not enabled.

Syntax
SYS(*cond*) *Instruction* [, *Rn*]
where:
cond is an optional condition code.
Instruction is the coprocessor instruction to execute.
Rn is an operand to the instruction. For instructions that take an argument, *Rn* is compulsory. For instructions that do not take an argument, *Rn* is optional and if it is not specified, *R0* is used. *Rn* must not be PC.

Usage
You can use this instruction to execute special coprocessor instructions such as cache, branch predictor, and TLB operations. The instructions operate by writing to special write-only coprocessor registers. The instruction names are the same as the write-only coprocessor register names and are listed in the *ARMv7-A Architecture Reference Manual*. For example:
SYS ICIALUES ; Invalidates all instruction caches Inner Shareable to Point of Unification and also flushes branch target cache.

Zürcher Fachhochschule

7

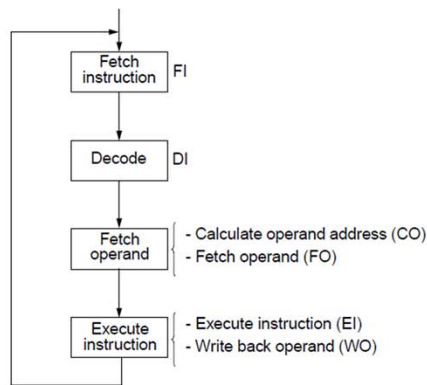
A classical Co-Processor is a device that has its own instruction set. The use of this device is triggered by co-processor instructions in the program code. One mechanism is as follows:

- 1.) The processor receives an Unidentified instruction
- 2.) The processor will (generally) ask its co-processors whether this instruction applies to them
- 3.) If no then it generates a unidentified op-code exception and lets normal exception processing take over
- 4.) Co-Processors will listen to the instruction and identify themselves
- 5.) The instruction is passed for processing to the co-processor.
- 6.) A processor will continue to execute instructions whilst the co-processor is doing its bit.
- 7.) If the co-processor is busy and the processor has a co-processor instruction next in its stream – the processor will stall
- 8.) A co-processor will generally have some form of exception registration

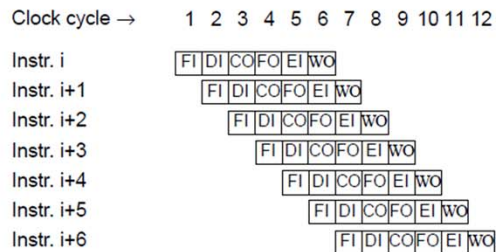
Loosely coupled co-processors have access to the same memory and external resources as the main processor through the system bus arbitration scheme. An older example is the 8087 numeric processor.

TCC and LCC are often found in video and sound streaming applications.

Instruction Parallelism (2) - Pipelines



FI: fetch instruction FO: fetch operand
DI: decode instruction EI: execute instruction
CO: calculate operand address WO: write operand



Execution time for the 7 instructions, with pipelining:
 $(T_{ex}/6) * 12 = 2 * T_{ex}$

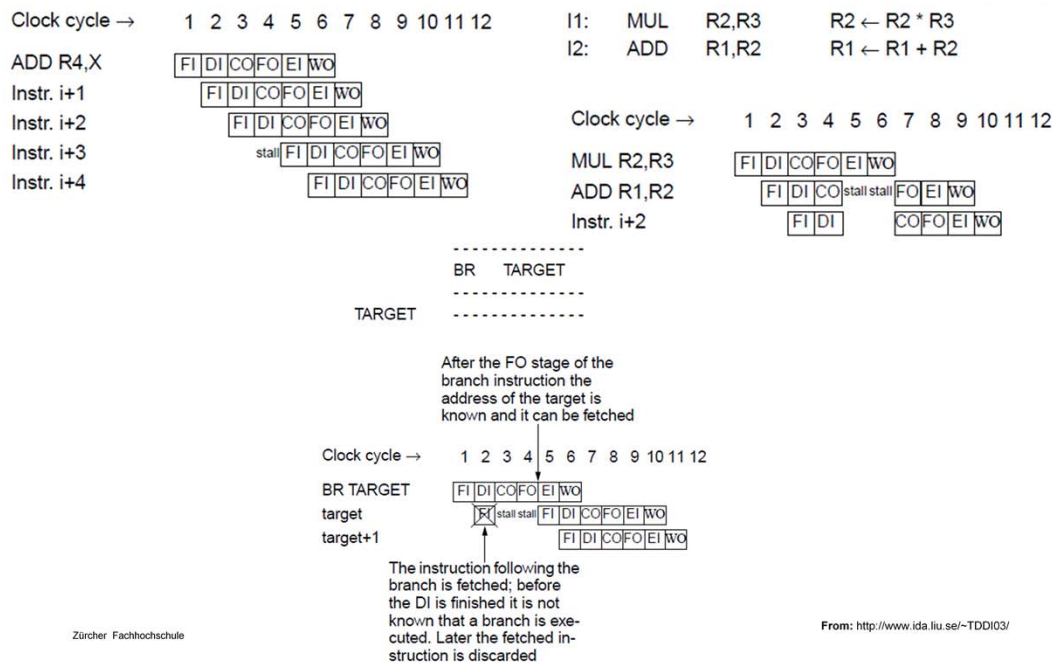
$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

From: <http://www.ida.liu.se/~TDDI03/>

Parallelism is the attempt to get things to work faster by throwing resources at it. The trick is to throw the right resources at the problem. There are various forms of parallelism: Instruction based parallelism is one, data parallelism is another, task parallelism a third. **Instruction Parallelism:** A pipeline is used in processor architectures to achieve instruction-level parallelism. In **Complex Instruction Set Computing (CISC)** the machine code abstracts from a series of operations the processor must perform. An add instruction in a typical CISC is actually three operations – loading the two registers into the accumulator, performing the addition and storing the result. In a CISC the instruction will be abstracted as ADD A,B with the result left in A. A and B could be registers. On the other hand A and B could equally be memory locations which need to be accessed before the addition can take place. Execution time therefore depends on where the data to be operated on is actually stored. Similarly the trade-off for fast compilers and smaller source code is increased complexity in the processor architecture itself. In a **Reduced Instruction Set Computing (RISC)** architecture a different approach is taken. The advantages are not obvious but – each instruction may be completed in one clock cycle. Also by delegating loading to one instruction and storing to another instruction and making sure the operations are kept separately, the circuitry for each instructions can be made much simpler. Therefore not only can the instruction be performed much faster, separate constant blocks of hardware are used for each instruction. Since an add requires operands that must be loaded the causality allows pipelining to be used to increase system throughput. The general operation is shown above. The net result is that one instruction is being executed every clock cycle – with a couple of exceptions – called hazards.

The difference thus between CISC and RISC is that CISC seeks to optimise the number of instructions per program at the cost of cycles per instruction whereas RISC seeks to optimise the cycles per instruction at the cost of instructions per program.

Instruction Parallelism (2)



Hazards are three-fold, structural hazards, data hazards and control hazards.

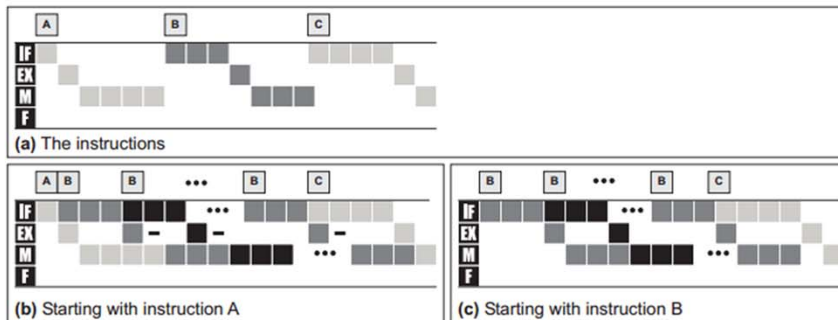
Structural hazards (top left) occur when a certain resource (memory, functional unit) is requested by more than one instruction at the same time. For instance if a memory is unable to accept another transfer with a clock cycle then the pipeline is stalled for a clock cycle. Certain resources can be duplicated in order to avoid structural hazards. Functional units (ALU, FP unit) can be pipelined themselves in order to support several instructions at a time. A classical way to avoid hazards at memory access is by providing separate data and instruction caches.

Data Hazards (top right) We have two instructions, I1 and I2. In a pipeline the execution of I2 can start before I1 has terminated. If in a certain stage of the pipeline, I2 needs the result produced by I1, but this result has not yet been generated, we have a data hazard. Some of the penalty produced by data hazards can be avoided using a technique called forwarding (bypassing).

Control hazards (bottom) are produced by branch instructions. Branch instructions represent a major problem in assuring an optimal flow through the pipeline. Several approaches have been taken for reducing branch penalties.

Pipelines – WCET Issues (1)

- Long Timing Effects – long term timing effects of instruction executions -> unbounded
- Caused if data dependencies between instruction I and $I + 1$
 - Negative – decrease in instruction time, 0 = no negative effect on WCET estimation



$$\begin{aligned}
 T(AB \dots BC) &= 7 + 3n \\
 T(AB \dots B) &= 6 + 3n \\
 T(B \dots BC) &= 6 + 3n \\
 T(B \dots B) &= 4 + 3n \\
 \delta_{A,C} &= 7 + 3n - 6 + 3n - 6 + 3n + 4 + 3n = -1
 \end{aligned}$$

From: J. Engblom. Processor Pipelines and Static Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, April 2002

10

Calculation is as follows:

Instruction A takes 6 cycles. B 7 cycles and C 6 cycles.

Instruction A-B takes 9 cycles, which is $9 - 6 - 7 = -4$ cycles (delta -> negative cycles saved)

Instruction B-C takes 9 cycles which is $9 - 7 - 6 = -4$ cycles

For the sequence in (b) -> $T(AB \dots BC) = 7 + 3n$ where n = number of B instructions (all of A + 1 of C)

$T(AB \dots B) = 6 + 3n$ (all of A + 3 times number of B's)

$T(B \dots BC) = 6 + 3n$ (start of B and end of C + 3 * number of B's)

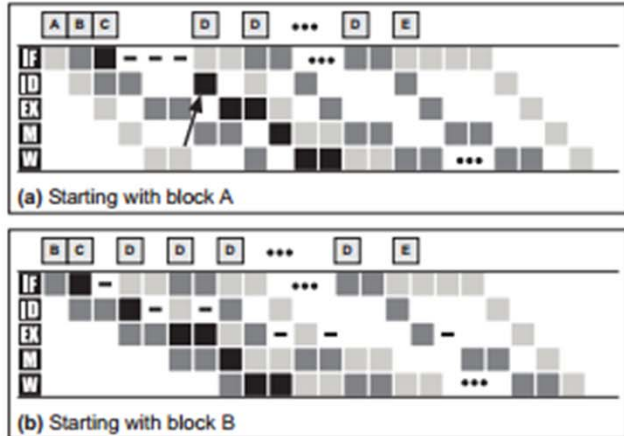
The delta = $7 + 3n - (6 + 3n) - (6 + 3n) + 4 + 3n = -1$

-1 is OK – your WCET estimation is still safe and tight.

Pipelines – WCET Issues (2)

- Positive – increase in actual processing time

- WCET estimation -> unsafe

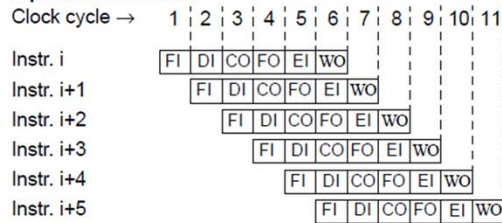


$$\begin{aligned}
 T(ABCD \dots DE) &= 14 + 2n \\
 T(ABCD \dots D) &= 12 + 2n \\
 T(BCD \dots DE) &= 11 + 2n \\
 T(BCD \dots D) &= 10 + 2n \\
 \delta_{A \dots C} &= 14 + 2n - 12 - 2n - 11 - 2n + 10 + 2n = +1
 \end{aligned}$$

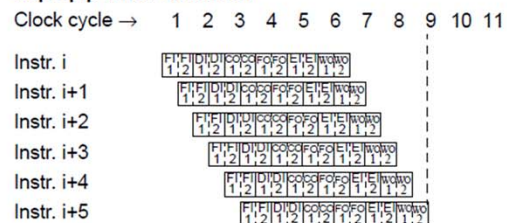
By performing n perfect instructions (NOP) long time effects can be removed

Superpipelined

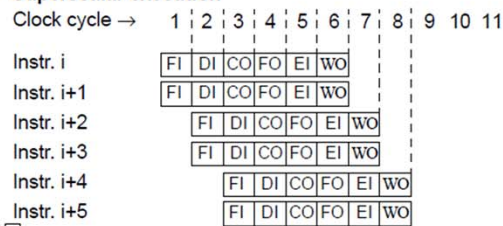
Pipelined execution



Superpipelined execution



Superscalar execution



Superpipelining is based on dividing the stages of a pipeline into substages and thus increasing the number of instructions which are supported by the pipeline at a given moment. By dividing each stage into two, the clock cycle period t will be reduced to the half, $t/2$; hence, at the maximum capacity, the pipeline produces a result every $t/2$ s. For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages; increasing the number of stages over this limit reduces the overall performance. **Superscalar** architectures allow several instructions to be issued and completed per clock cycle. A superscalar architecture consists of a number of pipelines that are working in parallel. Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel.

Three categories of limitations have to be considered:

Resource conflicts: They occur if two or more instructions compete for the same resource (register, memory, functional unit) at the same time; they are similar to structural hazards discussed with pipelines. Introducing several parallel pipelined units, superscalar architectures try to reduce risk of resource conflicts.

Control (procedural) dependency: The presence of branches creates major problems in assuring an optimal parallelism. If instructions are of variable length, they cannot be fetched and issued in parallel; an instruction has to be decoded in order to identify the following one and to fetch it so superscalar techniques are efficiently applicable to RISCs, with fixed instruction length and format.

Data conflicts: Data conflicts are produced by data dependencies between instructions in the program. Because superscalar architectures provide a great liberty in the order in which instructions can be issued and completed, data dependencies have to be considered with much attention. So we now see that the issue becomes one of scheduling **True data dependency** exists when the output of one instruction is required as an input to a subsequent instruction; An **output dependency** exists if two instructions are writing into the same location; if the second instruction writes before the first one, an error occurs; ...

Superscalar

I1: ADDF R12,R13,R14 R12 ← R13 + R14 (float. pnt.)
 I2: ADD R1,R8,R9 R1 ← R8 + R9
 I3: MUL R4,R2,R3 R4 ← R2 * R3
 I4: MUL R5,R6,R7 R5 ← R6 * R7
 I5: ADD R10,R5,R7 R10 ← R5 + R7
 I6: ADD R11,R2,R3 R11 ← R2 + R3

Decode/ Issue	Execute	Writeback/ Complete	Cycle
I1 I2			1
I3 I4	I1 I2		2
I5 I6	I1		3
		I3 I2	4
		I3	5
	I5	I4	6
	I6	I5	7
		I6	8

Decode/ Issue	Execute	Writeback/ Complete	Cycle
I1 I2			1
I6 I4	I1 I2		2
I5 I3	I1		3
	I6 I4	I1 I2	4
	I5 I3	I6 I4	5
		I5 I3	6
			7
			8

Decode/ Issue	Execute	Writeback/ Complete	Cycle
I1 I2			1
I3 I4	I1 I2		2
I5 I6	I1 I3	I2	3
	I6 I4	I1 I3	4
	I5	I4 I6	5
		I5	6
			7
			8

Zürcher Fachhochschule

From: <http://www.ida.liu.se/~TDDI03/>

13

... An **antidependency** exists if an instruction uses a location as an operand while a following one is writing into that location; if the first one is still using the location when the second one writes into it, an error occurs

These conflicts are to be managed when we look at the policies used for instruction execution which are characterized by the following two factors: the order in which instructions are issued for execution; the order in which instructions are completed (they write results into registers and memory locations). The simplest policy is to execute and complete instructions in their sequential order. This, however, gives little chances to find instructions which can be executed in parallel. In order to improve parallelism the processor has to look ahead and try to find independent instructions to execute in parallel. Instructions will be executed in an order different from the strictly sequential one, with the restriction that the result must be correct.

Execution policies:

1. In-order issue with in-order completion. (upper right)

Instructions are issued in the exact order that would correspond to sequential execution; results are written (completion) in the same order. An instruction cannot be issued before the previous one has been issued; An instruction completes only after the previous one has completed.

2. In-order issue with out-of-order completion. (lower right)

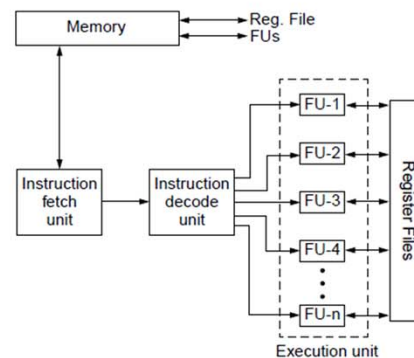
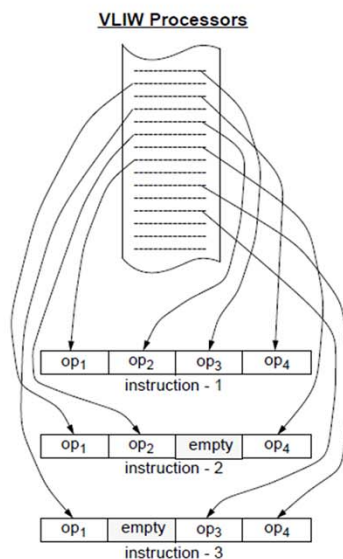
With in-order issue, no new instruction can be issued when the processor has detected a conflict and is stalled, until after the conflict has been resolved.

3. Out-of-order issue with out-of-order completion. (lower left)

With out-of-order issue & out-of-order completion the processor has to bother about true data dependency and both about output-dependency and antidependency!

With superscalar processors we are interested in techniques which are not compiler based but allow the hardware alone to detect instructions which can be executed in parallel and to issue them.

VLIW Processors



Zürcher Fachhochschule

From: <http://www.ida.liu.se/~TDDI03/>

14

Very Large Instruction Word VLIW architectures rely on **compile-time** detection of parallelism. The compiler analyses the program and detects operations that can be executed in parallel (on the specific hardware); such operations are packed into one “large” instruction. After one instruction has been fetched all the corresponding operations are performed in parallel. Since it is a compiler function, no hardware is needed for run-time detection of the parallelism – the compiler is responsible for scheduling and can achieve this over the entire run-time of the program – off-line. In other words hardware complexity – a unit-for-unit expense - is traded for compiler complexity.

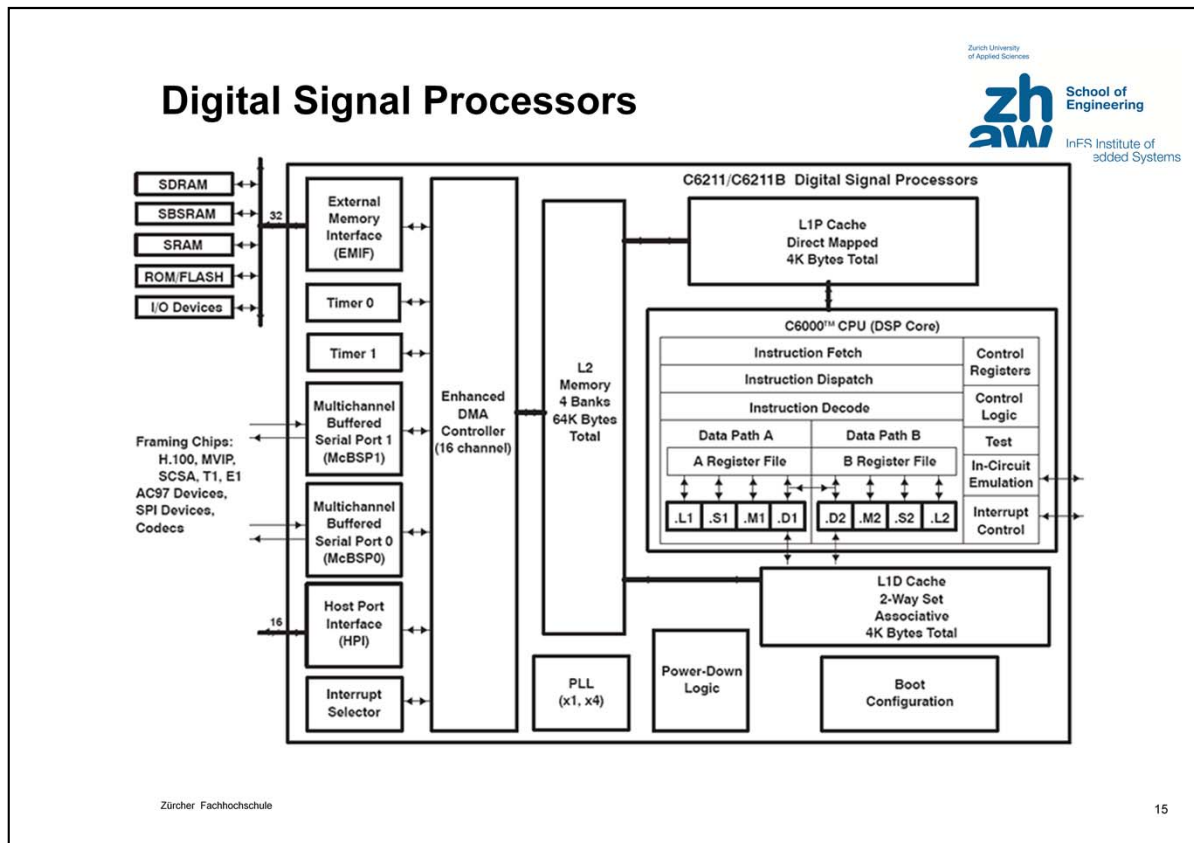
Advantages: Simple hardware: the number of FUs can be increased without needing additional sophisticated hardware to detect parallelism, unlike superscalar architectures. Good compilers can detect parallelism based on global analysis of the whole program (no window of execution problem).

Problems: Large number of registers needed in order to keep all FUs active (to store operands and results). Large data transport capacity is needed between both FUs and the register file and between register files and memory. High bandwidth required between instruction cache and fetch unit. Example: one instruction with 7 operations, each 24 bits -> 168 bits/instruction. Large code size, partially because unused operations cause wasted bits in instruction word. Incompatibility of binary code between processor versions

For example: If for a new version of the processor additional FU's are introduced the number of operations possible to execute in parallel is increased the instruction word changes old binary code cannot be run on this processor.

A good introduction is given by:

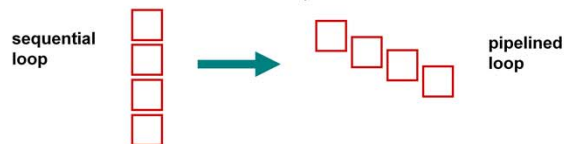
psut.edu.jo/sites/%2Fqaralleh%2FCOE%2FCA%2520Documents%2F11_vliw.pdf



Digital Signal Processors are the ultimate application specific machine and as such have benefited from many advanced concepts in architectures. DSP's are designed for high speed numeric calculations and as such stretch the limits of processor design. The block diagram above is a fairly typical example of a modern DSP. The core task is things like filters on streaming data which ends up, more often than not, a convolution operation which reduces to a multiplication and addition. So the core of a Functional Unit is a Multiply and Accumulate unit (.M) there are also several ALU's in this unit. The .D unit is for data accesses the .L (ALU) and .S (Shifter/ALU) units handle general arithmetic, logical and branch operations. The DSP has a modified **Harvard architecture** – in this case the physical memory bus is shared but the use of cache allows the separation of data and code accesses. Most high performance DSP's will have separate pins for data and code data busses. There are two of these functional units -> and it's a VLIW architecture (256-bit wide instruction). Similarly it has the load store architecture reminiscent of RISC processors. DSP's are not, as a rule, compiler friendly and over 50% of DSP solutions are programmed on naked hardware in assembler. Interestingly enough the BIOS – a sort of OS from Texas Instruments is something like the 8th most popular OS in embedded systems (www.embedded.com).

Software Pipelining

- DSP and compilers
 - A difficult history
 - Most code in loops but loops sometimes need to be constructed that compiler can efficiently schedule loop execution
- Software Pipelining is the technique of scheduling instructions across several iterations of a loop.
 - reduces pipeline stalls on sequential pipelined machines
 - exploits instruction level parallelism on superscalar and VLIW machines
 - iterations are overlaid so that an iteration starts before the previous iteration have completed



Zürcher Fachhochschule

This and the next three slides 100% lifted from cs.nyu.edu/courses/spring02/G22.3130-001/vliw_epic_4_2002.ppt

Software Pipelining Example


- Source code:

```
for(i=0; i<n; i++) sum += a[i];
```

- Loop body in assembly:

```
r1 = L r0  
--- ;stall  
r2 = Add r2,r1  
r0 = add r0,4
```

- Unroll loop &
allocate registers



```
r1 = L r0  
--- ;stall  
r2 = Add r2,r1  
r0 = Add r0,12  
r4 = L r3  
--- ;stall  
r2 = Add r2,r4  
r3 = add r3,12  
r7 = L r6  
--- ;stall  
r2 = Add r2,r7  
r6 = add r6,12  
r10 = L r9  
--- ;stall  
r2 = Add r2,r10  
r9 = add r9,12
```

Software Pipelining Example (cont)

Schedule Unrolled Instructions, exploiting VLIW (or not)

```
r1 = L r0
r4 = L r3
r2 = Add r2,r1 r7 = L r6
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9
r3 = add r3,12 r2 = Add r2,r7 r1 = L r0
r6 = add r6,12 r2 = Add r2,r10 r4 = L r3
r9 = add r9,12 r2 = Add r2,r1 r7 = L r6
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9
r3 = add r3,12 r2 = Add r2,r7 r1 = L r0
r6 = add r6,12 r2 = Add r2,r10 r4 = L r3
r9 = add r9,12 r2 = Add r2,r1 r7 = L r6
. . .
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9
r3 = add r3,12 r2 = Add r2,r7
r6 = add r6,12 Add r2,r10
r9 = add r9,12
```

Identify
repeating
pattern
(kernel)

Zürcher Fachhochschule

Software Pipelining Example (cont)

Loop becomes:

```
r1 = L r0  
r4 = L r3  
r2 = Add r2,r1 r7 = L r6
```

← prolog

```
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9  
r3 = Add r3,12 r2 = Add r2,r7 r1 = L r0  
r6 = Add r6,12 r2 = Add r2,r10 r4 = L r3  
r9 = Add r9,12 r2 = Add r2,r1 r7 = L r6
```

← kernel

```
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9  
r3 = Add r3,12 r2 = Add r2,r7  
r6 = Add r6,12 Add r2,r10  
r9 = Add r9,12
```

← epilog

«High Level» Software Pipelining

int mac(const short *a, const short *b, int sqr, int *sum)

```

{
  int i;
  int dotp = *sum;
  for(i=0; i<150; i++)
  {
    dotp+=b[i]*a[i];
    sqr+=b[i]*b[i];
  }
  *sum=dotp;
  return sqr;
}

```

(1) Original C Code

int i,x,y;
int dotp = *sum;
x=a[0];
y=b[0];
for(i=1; i<150; i++)
{
 dotp+=y *x;
 sqr+=y * y;
 x=a[i];
 y=b[i];
}
dotp+= y * x;
sqr+= y * y;
*sum=dotp;
return sqr;
}

(2) Improved C Code

move v:(r0),x0
move y:(r4)+,y0
move y:(r6+3),a
asr #23,a
mac +x0,y0,a y:(r0),x0
asl #23,a
v:(r0)+,y0
move a1,y:(r6+3) y:(r6+4),a
asr #23,a
mac +x0,y0,a
asl #23,a
move a1,y:(r6+4)

asr #23,a
mac +x0,y0,a
asl #23,a
asr #23,b b y:(r0)+,y0
mac +x0,x0,b y:(r4)+,x0
asl #23,b,b

(2) Innermost loop body from original C code

(4) Innermost loop body from improved C code

- (DSP) Compilers perform pipelining
 - By examining assembler code the quality of compiler generated pipelining can be determined
 - The loop can be re-structured in C
 - With this iterative process an optimal code can be generated

Figure 4 Dot Product Program

SW Pipelined VLIW

- Example loop kernel for array multiplication on TI DSP
- Two VLIW Instructions
- Loops pipelined

```

L9:
|| B .S2
|| LDW .D2
|| LDW .D1
|| STW .D1
|| MFYSP .M1X
|| SUM .L2

L9
; @@
; B5++, B4
; A3++, A0
;
;
; A5, *A4++
; B4, A0, A5
; B0, 1, B0
; @@
; @@

```

Executed in parallel

Iteration being performed (@@ =
n+2)

Pipelining Functional Units

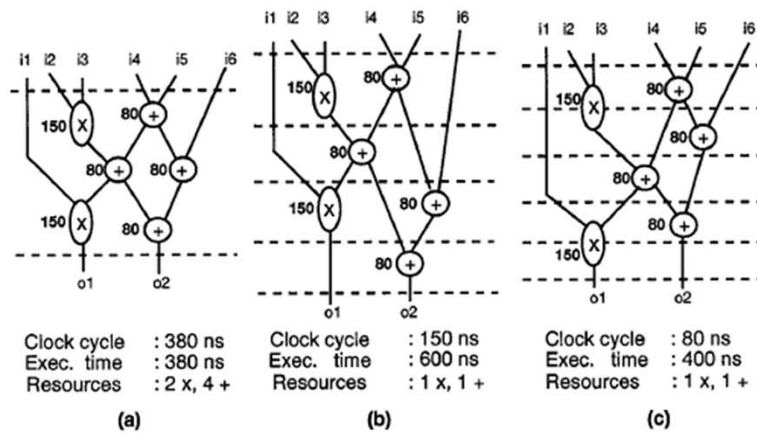


Fig. 1. Effect of clock period on execution time and resources required.

En-shou Chang, Daniel D. Gajski, "An Optimal Clock Period Selection Method Based on Slack Minimization Criteria" ACM Trans. Design Automation of Electronic Systems, Vol.1, No.3 (1996)

In the above slide, on the far left, we see a data flow graph with 6 operations, 2 MUL's and 4 adds. Beside each operation is the time it takes to execute. The task of scheduling can be said to be to divide the behavioural description into discrete control steps, usually one cycle long, consisting of fetching data from a register, transformed by a functional unit and written back to a register. In our first example this process takes, measured by the longest path within the functional unit (consisting of 4 adders and 2 multipliers), 380ns. If we partition this design into 4 steps of 150 ns each we see the execution time increases to 600ns but the number of necessary FU's decreases to one multiplier and one adder. Finally by reducing the clock cycle to 80ns we get a total time of 400ns and again 1 MUL and 1 ADD which would appear to be a pretty optimal design in terms of latency versus resources (i.e. cost) – at the expense of power (increased clock frequency).

Note that until now we have followed the path allocation -> binding -> scheduling. In this particular design case the line is scheduling -> allocation -> binding

Pipelining Designs

Zürich University
of Applied Sciences



School of
Engineering

InES Institute of
Embedded Systems

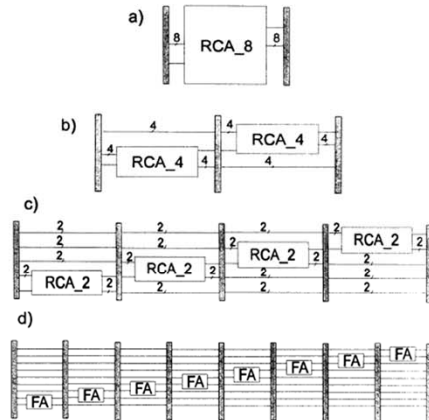
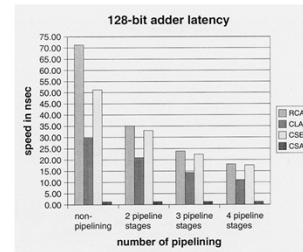
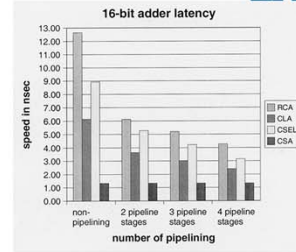


Figure 11: Pipelined RCA to a) $\beta=8$; b) $\beta=4$; c) $\beta=2$; and d) $\beta=1$.



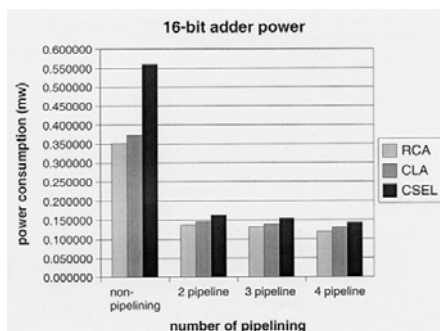
Zürcher Fachhochschule

23

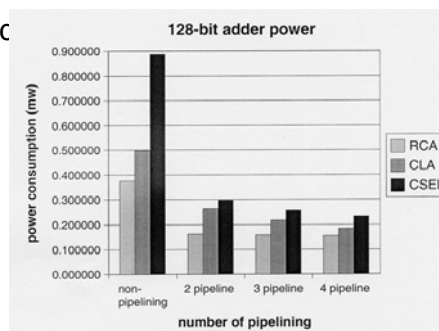
The adder is a basic block in many arithmetic designs. In HW/SW applications one limiting factor – in other words a reason to design one's own adder - of the addition speed of the ALU is when bit sizes larger than that of the ALU are required. Adders are not perfect and there is a wealth of literature associated with their design – for simplicities sake we take the Ripple Carry Adder (RCA) as an example. The limiting factor in the RCA is the carry signal which in an 8-bit adder ripples from one adder to another. This is a well known effect and several architectures have been developed to compensate for this namely Carry Look-Ahead (CLA) Carry Select Adder (CSEL) and Carry Save Adder (CSA). There are a number of publications dealing with pipelining adders and the results are very platform (i.e. FPGA and Library) specific. The picture on the left [1] shows the principle of pipelining a full adder and the authors, with their platform, calculate that the 2 stage 4-bit pipeline is the fastest. Experiments carried out by [2] are shown above right and the authors clearly achieve efficiencies with higher pipeline stages. The power consumption was also measured and is shown below:

[1] Peiro M.M., Coquillar, J.V., Boemo, E. "On the Design of FPGA-Based Multioperand Pipeline Adders"

[2] Wei, L., "Implementation of Pipelined Bit-parallel Adders"



Adc



Pipelining Designs

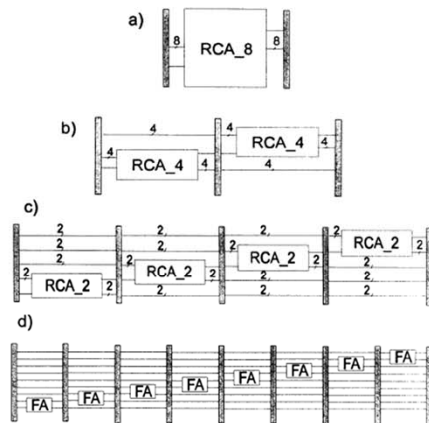
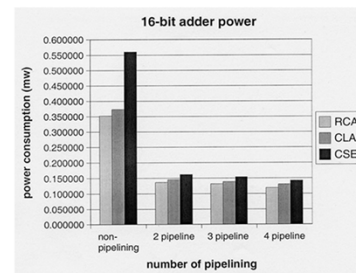
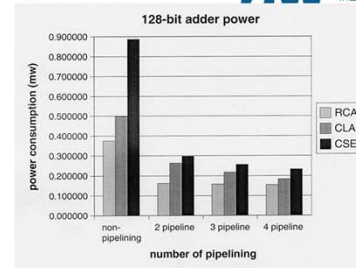


Figure 11: Pipelined RCA to a) $\beta=8$; b) $\beta=4$; c) $\beta=2$; and d) $\beta=1$.



The adder is a basic block in many arithmetic designs. In HW/SW applications one limiting factor – in other words a reason to design one's own adder - of the addition speed of the ALU is when bit sizes larger than that of the ALU are required. Adders are not perfect and there is a wealth of literature associated with their design – for simplicities sake we take the Ripple Carry Adder (RCA) as an example. The limiting factor in the RCA is the carry signal which in an 8-bit adder ripples from one adder to another. This is a well known effect and several architectures have been developed to compensate for this namely Carry Look-Ahead (CLA) Carry Select Adder (CSEL) and Carry Save Adder (CSA). There are a number of publications dealing with pipelining adders and the results are very platform (i.e. FPGA and Library) specific. The picture on the left [1] shows the principle of pipelining a full adder and the authors, with their platform, calculate that the 2 stage 4-bit pipeline is the fastest. Experiments carried out by [2] are shown above right and the authors clearly achieve efficiencies with higher pipeline stages. The power consumption was also measured and is shown above:

[1] Peiro M.M., Coquillar, J.V., Boemo, E. "On the Design of FPGA-Based Multioperand Pipeline Adders"

[2] Wei, L., "Implementation of Pipelined Bit-parallel Adders"

[3] Dadda L., Piuri, V. "Pipelined Adders"