CS420/520
Computer Architecture I

Instruction-Level Parallelism

Dr. Xiaobo Zhou
Department of Computer Science

Re: Pipeline Data Hazard Detection (Delay Load)

Situation	Example code sequence	Action
No dependence	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because no dependence exists on R1 in the immediately following three instructions.
Dependence requiring stall	LD **R1**,45(R2) DADD R5,**R1**,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.
Dependence overcome by forwarding	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,**R1**,R7 OR R9,R6,R7	Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.
Dependence with accesses in order	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,**R1**,R7	No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

° **Comparing the destination and sources of adjacent instructions**

What is Instruction-level Parallelism (ILP)

- ILP: the potential overlap among instruction executions due to pipelining
  - The instructions can be executed and evaluated in parallel
- How to exploit ILP
  - Hardware stall
  - Software NOP
  - Hardware forwarding/bypassing
- Two MORE largely separable approaches to exploiting ILP
  - Static/compiler pipeline scheduling by the compiler tries to minimize stalls by separating dependent instructions so that they will not lead to hazards
  - Dynamic hardware scheduling tries to avoid stalls when dependences, which could generate hazards, are present.

Software (compiler) Static Scheduling / ILP

- Software scheduling: the goal is to exploit ILP by preserving program order only where it *affects the outcome of the program*

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d, e, and f in memory.

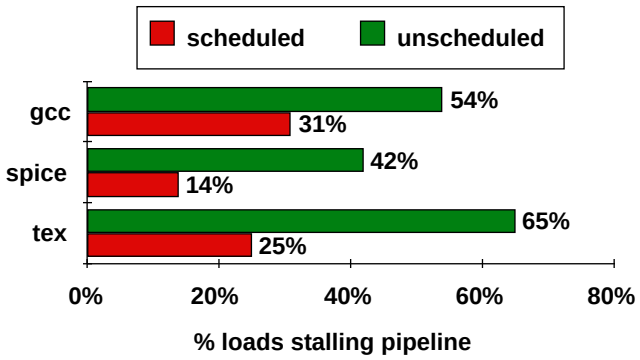
Slow code:

```
LW   Rb,b
LW   Rc,c
ADD  Ra,Rb,Rc
SW   a,Ra
LW   Re,e
LW   Rf,f
SUB  Rd,Re,Rf
SW   d,Rd
```

Fast code:

```
LW   Rb,b
LW   Rc,c
LW   Re,e
ADD  Ra,Rb,Rc
LW   Rf,f
SW   a,Ra
SUB  Rd,Re,Rf
SW   d,Rd
```

Compiler Avoiding Load Stalls:



Data Dependences

° Data dependence

- Instruction *i* produces a result that may be used by instruction *j*
- Instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i (a chain of dependences)*

```
loop:
        LD      F0, 0(R1)
        DADD    F4, F0, F2
        SD      F4, 0(R1)
        DAADI   R1, R1, -8
        BNE     R1, R2, Loop
```

Name Dependences

° Name dependence (not-true-data-hazard)

- Occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associating with that name
- Remember: do not be restricted to the 5-stage pipeline!

Anti-dependence (WAR)

j writes a register or memory location that *i* reads:

```
ADD   $1, $2, $4
SUB   $4, $5, $6
```

What if SUB does earlier than ADD?
Is there a data flow?

Output dependence (WAW)

i and *j* write the same register or memory location

```
SUB   $4, $2, $7
SUBI  $4, $5, 100
```

What if SUBI does earlier than SUB?
Is there a data flow?

How many ways for data to flow between instructions?

Data Hazards - RAW

° Data hazards may be classified, depending on the order of read and write accesses in the instructions

° RAR (read after read) is not a hazard, nor a name dependence

° RAW (read after write):

- *j* tries to read a source before *i* writes it, so *j* incorrectly gets the old value; most common type – true data hazards

Example?

Data Hazards - WAW

° WAW (write after write):

- Output dependence of name hazards: *j* tries to write an operand before it is written by *i*.

Can you nominate an example?

Short/long pipelines

```
MULTF  F4, F5, F6
LD F4, 0(F1)
```

Is WAW possible in the MIPS classic five-stage integer pipelining? Why?

Data Hazards - WAR

° WAR (write after read):

- Anti-dependence of name hazards: *j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the new value

Example?

1) Due to re-ordering

```
DIV  $0, $2, $4      (add is stalled waiting DIV,
ADD $6, $0, $8       if SUB is done before ADD,
SUB $8, $10, $14     anti-dependence violated).
```

2) One writes early in the pipeline and some others read a source late in the pipeline

Is WAR possible in the MIPS classic five-stage integer pipelining?

ILP and Dynamic Scheduling

° Dynamic scheduling: the goal is to exploit ILP by preserving program order only where it *affects the outcome of the program*

Out-of-order execution

```
DDIV   F0, F2, F4
DADD   F10, F0, F8
DSUB   F12, F8, F14   // DSUB not dependent on
                      // anything in the pipeline
                      // can its order be exchanged
                      // with DADD?
```

Out-of-order execution may introduce WAR and WAW hazards

```
DDIV   F0, F2, F4
DADD   F6, F0, F8     // anti-dependence between DADD
DSUB   F8, F10, F14   // and DSUB; if out-of-order, WAR
DMUL   F6, F10, F8    // register renaming helps!
```


Dynamic Scheduling – Tomasulo' Register Renaming

Before:

```
DDIV   F0, F2, F4
DADD   F6, F0, F8
SD     F6, 0(R1)
DSUB   F8, F10, F14
DMUL   F6, F10, F8
```

//anti-dependence DSUB – F8, WAR
//output dependence DMUL-F6, WAW

// How many true data dependences?

After:

```
DDIV   F0, F2, F4
DADD   S, F0, F8
SD     S, 0(R1)
DSUB   T, F10, F14
DMUL   F6, F10, T
```

What are dependencies there?

What dependencies disappear? And what are still there?

What to do with the subsequence use of F8?

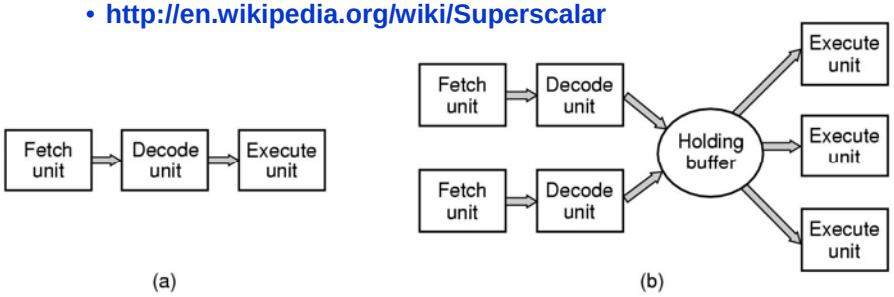
Finding the subsequent use of F8 requires compiler analysis or hardware support

Concurrency and Parallelism

- Concurrency in software is a way to manage the sharing of resources efficiently at the same time
  - When multiple software threads of execution are running concurrently, the execution of the threads is interleaved onto a single hardware resource
    - Why not schedule another thread while one thread is on a cache miss or even page fault?
    - Time-sliced multi-threading increases concurrency, overlapping CPU time with I/O time
    - Pseudo-parallelism
- True parallelism requires multiple hardware resources
  - Multiple software threads are running simultaneously on different hardware resources/processing elements
  - One approach to addressing thread-level true parallelism is to increase the number of physical processors / execution elements in the computer

ILP and Super-Scalar Processors

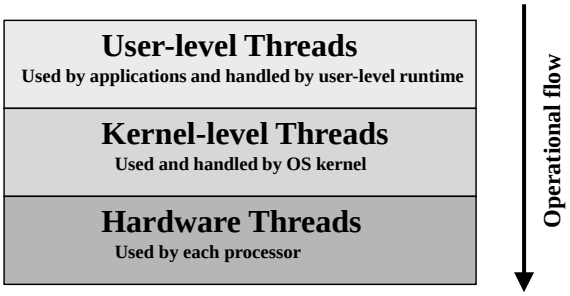
- ILP: the potential overlap among instruction executions due to pipelining
  - To increase the number of instructions that are executed by the processor on a single clock cycle
  - A processor that is capable of executing multiple instructions *in a single clock cycle* is known as a super-scalar processor
  - http://en.wikipedia.org/wiki/Superscalar



(a) A three-stage path (b) A superscalar CPU

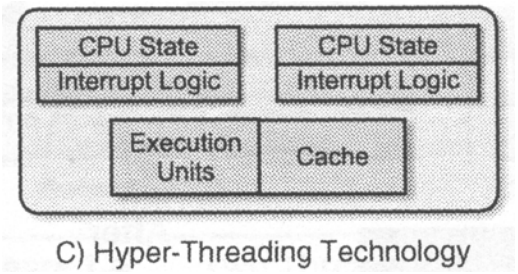
Multi-Threading

- **Process: for resource grouping and execution**
- **Thread: a finer-granularity entity for execution and parallelism**
  - **Lightweight processes, multithreading**
- **Multi-threading: Operating system supports multiple threads of execution within a single process**
  - **At hardware level, a thread is an execution path that remains independent of other hardware thread execution paths**



Simultaneous Multi-threading (Hyper-Threading)

- **Simultaneous multi-threading (SMT)**
  - Create multiple logical processors with a physical processor
  - One logical processor for a thread, which requires an architecture state consisting of the GPRs and Interrupt logic
  - Duplicate multiple architecture states (CPU state), and, let other CPU resources, such as caches, buses, execution units, branch prediction logic shared among architecture states
  - Multi-threading at hardware level, instead of OS switching
  - Intel's SMT implementation is called Hyper-Threading Technology (HT Technology)



What is next?

Multi-Core

Processor performance

- Moore's Law
- Decreasing transistor sizes
- Increasing number of transistors
- Decreasing chip sizes

↓

Higher clock rates
More complex
Faster microprocessors

↓

However

Transistors can't shrink forever
Several problems with this approach
Performance increase is slowing down → SOLUTION MULTICORE PROCESSORS
1990-2000 : 60% per year
2000-2004 : 40% per year

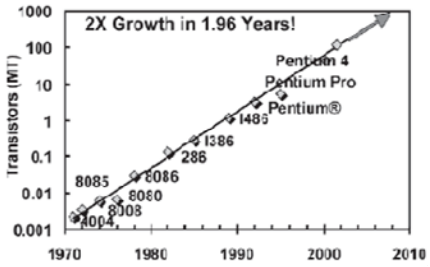


Fig. 1 Moore's Law: Transistors on a chip double every 2 years(Gelsinger, 2001)

Limitations of Sing-core Technology

a) Power consumption

Smaller gates
Higher clock rates
2-3% power increase per 1% performance increase
Leakage power scales 5X in each technology node

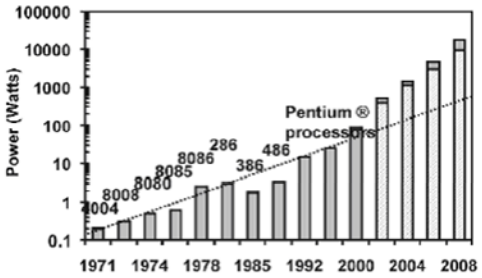


Fig. 2 Lead microprocessor power increases beyond expected(Gelsinger, 2001)

b) Heat generation

Extensive heat rejection of multiple kilowatts is undesirable

c) On-chip wires and interconnect latency

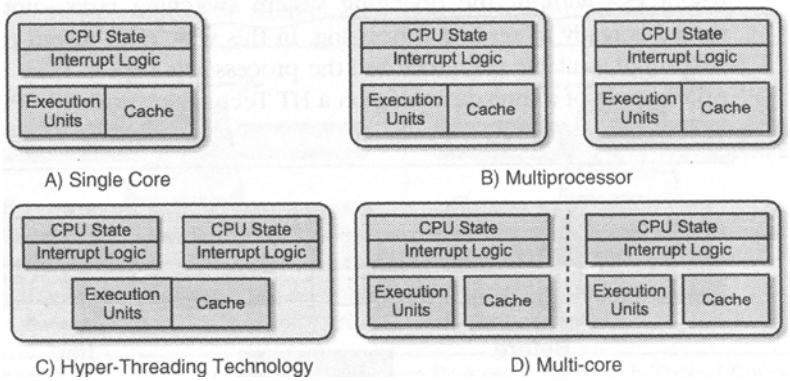
Faster gates and larger chips
On-chip wire delay increases
Interconnect delays become more critical

d) Limited parallelism

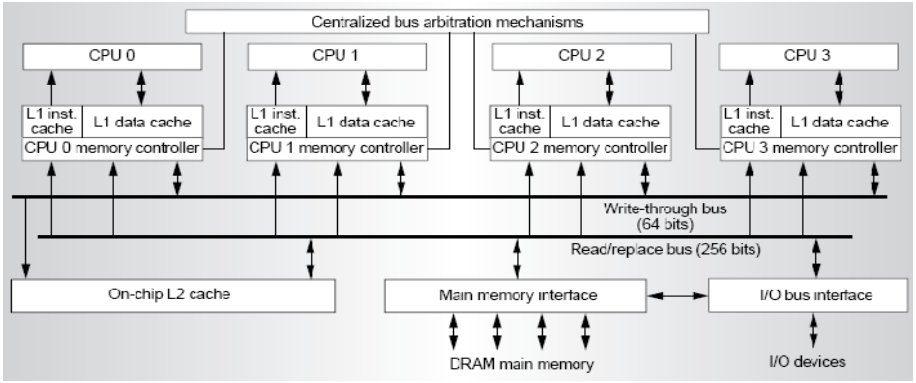
Finite amount of ILP
Interdependent instructions

Multi-Processor and Multi-Core

° multi-core processors use chip multi-processing (CMP)
- Cores are essentially two individual processors on a single die
- May or may not share on-chip cache
- True parallelism, instead of high concurrency



Stanford HYDRA Multi-core Structure



Multi-core Processors as a Solution

a) Advantages and improvements

- Performance increase by means of parallel execution
- Power and Energy efficient cores (Dynamic power coordination)
- Minimized wire lengths and interconnect latencies
- Both ILP and thread level parallelism
- Reduced design time and complexity

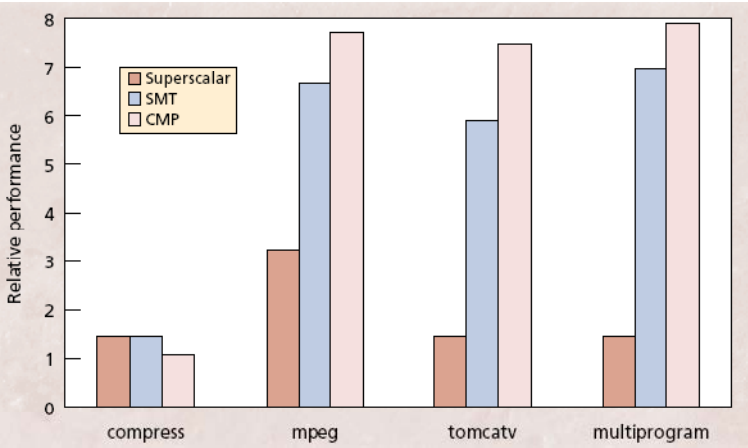
b) Task management and parceling

- Usually operating system distributes the application to the cores
- Can be done according to resource requirements

c) Multicore examples

AMD Opteron Enterprise, Athlon64
Intel Pentium Extreme edition, Pentium D
IBM Power 4, Power 5 and Sun Niagara

Multi-core vs Superscalar vs SMT



Multicore VS Superscalar VS SMT(Hammond, 1997)

Multi-core vs Superscalar

• Superscalar processors differ from multi-core processors in that the redundant functional units are not entire processors.

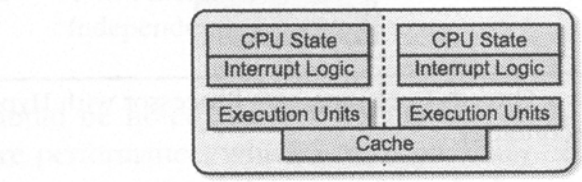
  • A single superscalar processor is composed of finer-grained functional units such as the ALU, integer shifter, etc. There may be multiple versions of each functional unit to enable execution of many instructions in parallel. This differs from a multicore CPU that concurrently processes instructions from multiple threads, one thread per core.

•The various techniques are not mutually exclusive—they can be (and frequently are) combined in a single processor. Thus a multicore CPU is possible where each core is an independent processor containing multiple parallel pipelines, each pipeline being superscalar.

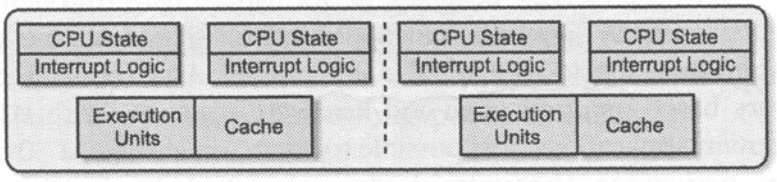
http://en.wikipedia.org/wiki/Superscalar

Multi-Core and SMT

° multi-core processors can be combined with SMT technology



E) Multi-core with Shared Cache



F) Multi-core with Hyper-Threading Technology

Pthreads Overview

- What are Pthreads?
  - An IEEE standardized thread programming interface
  - POSIX threads
  - Defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library

To software developer:

a thread is a "procedure" that runs independently from its main program

The Pthreads API

- The API is defined in the ANSI/IEEE POSIX 1003.1 – 1995
  - Naming conventions: all identifiers in the library begins with pthread_
  - Three major classes of subroutines
    - Thread management, mutexes, condition variables

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

Multi-Thread Programming Reference Links

- http://www.llnl.gov/computing/tutorials/pthreads/
  - Many examples and examples with bugs
- http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- http://www.gnu.org/software/libc/manual/html_node/POSIX-Threads.html
- GDB for debugging:
  http://sources.redhat.com/gdb/current/onlinedocs/gdb_6.html
- DDD for debugging:

  http://www.gnu.org/manual/ddd/html_mono/ddd.html

Where to get more information?

- CA4
  - 2.1, 2.4 (pages 89 – 93)
- CA3:
  - 3.1 – 3.2, 4.1
- Multi-core programming, by Shameem Akhter and Jason Roberts, Intel Press (ISBN 0-9764832-4-6)
- Internet
  - Find out more about Multi-core technology, use your own words and write a summary report from both the hardware support and software programming viewpoints

Homework

- Pipeline homework; see course Web site.
- Reading assignment; see course Web site.

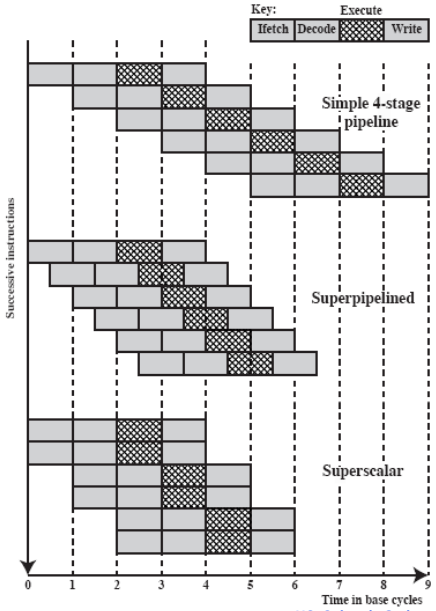
How to Improve Concurrency and Parallelism

- One approach to address the increasingly concurrent nature of modern software involves using a pre-emptive, or time-sliced, multitasking operating system
  - Time-sliced multi-threading increases concurrency, overlapping CPU time with I/O time
- One approach to address thread-level true parallelism is to increase the number of physical processors / execution elements in the computer

Concurrency and Parallelism

- Concurrency in software is a way to manage the sharing of resources efficiently at the same time
  - When multiple software threads of execution are running concurrently, the execution of the threads is interleaved onto a single hardware resource
    - Why not schedule another thread while one thread is on a cache miss or even page fault?
    - Pseudo-parallelism
- True parallelism requires multiple hardware resources
  - Multiple software threads are running simultaneously on different hardware resources/processing elements

Superscalar and Superpipelining



- Superpipelining exploits the fact that many pipeline stages perform tasks that require less than half a clock cycle.
  - Thus, a doubled internal clock speed allows the performance of two tasks in one external clock cycle [Stallings Com. O&A 7e]
- Superscalar [Shen & Lipasti, McGraw Hill]