CS 211: Computer Architecture

Introduction to ILP Processors & Concepts

Course Outline

- Introduction: Trends, Performance models
- Review of computer organization and ISA implementation
- Overview of Pipelining
- ILP Processors: Superscalar Processors
  - Next! ILP Intro and Superscalar
- ILP: EPIC/VLIW Processors
- Compiler optimization techniques for ILP processors – getting max performance out of ILP design
- Part 2: Other components- memory, I/O.

Introduction to Instruction Level Parallelism (ILP)

- What is ILP?
  - Processor and Compiler design techniques that speed up execution by causing individual machine operations to execute in parallel
- ILP is transparent to the user
  - Multiple operations executed in parallel even though the system is handed a single program written with a sequential processor in mind
- Same execution hardware as a normal RISC machine
  - May be more than one of any given type of hardware

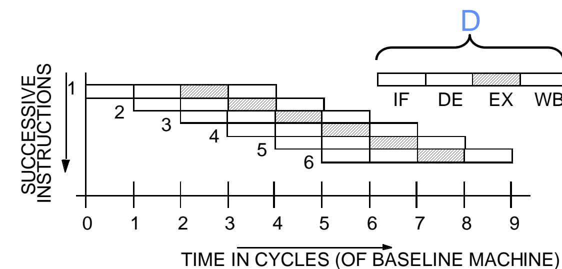
Architectures for Instruction-Level Parallelism

Scalar Pipeline (baseline)

Instruction Parallelism = D

Operation Latency = 1

Peak IPC = 1



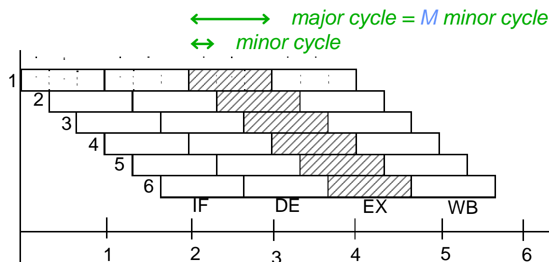
Superpipelined Machine

Superpipelined Execution

IP = *DxM*

OL = *M minor cycles*

Peak IPC = *1 per minor cycle (M per baseline cycle)*



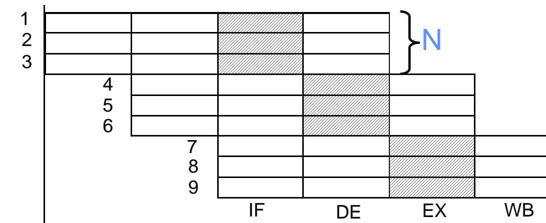
Superscalar Machines

Superscalar (Pipelined) Execution

IP = *DxN*

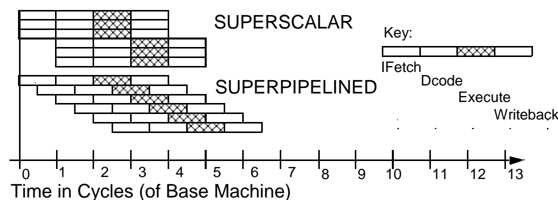
OL = *1 baseline cycles*

Peak IPC = *N per baseline cycle*



Superscalar and Superpipelined

<u>Superscalar Parallelism</u>	<u>Superpipeline Parallelism</u>
Operation Latency: 1	Operation Latency: M
Issuing Rate: N	Issuing Rate: 1
Superscalar Degree (SSD): N	Superpipelined Degree (SPD): M
(Determined by Issue Rate)	(Determined by Operation Latency)



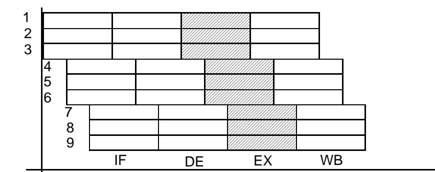
Superscalar and superpipelined machines of equal degree have roughly the same performance, i.e. if n = m then both have about the same IPC.

Limitations of Inorder Pipelines

- **CPI of inorder pipelines degrades very sharply if the machine parallelism is increased beyond a certain point,** *i.e. when NxM approaches average distance between dependent instructions*
- **Forwarding is no longer effective**

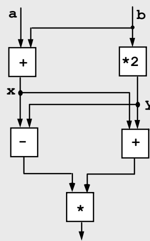
  ⇒ must stall more often

  – *Pipeline may never be full due to frequent dependency stalls!!*



What is parallelism and where

```
x = a + b;
y = b * 2
z =(x-y) * (x+y)
```



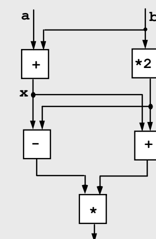
What is Parallelism?

- **Work**
  - T_1 - time to complete a computation on a sequential system
- **Critical Path**
  - T_∞ - time to complete the same computation on an infinitely-parallel system
- **Average Parallelism**
  - $P_{avg} = T_1 / T_\infty$
- **For a p wide system**

$$T_p \geq \max\{ T_1/p, T_\infty \}$$

$$P_{avg} >> p \Rightarrow T_p \approx T_1/p$$

```
x = a + b;
y = b * 2
z =(x-y) * (x+y)
```



Example Execution

Functional Unit	Operations Performed	Latency
Integer Unit 1	Integer ALU Operations Integer Multiplication Loads Stores	1 2 2 1
Integer Unit 2 / Branch Unit	Integer ALU Operations Integer Multiplication Loads Stores Test-and-branch	1 2 2 1 1
Floating-point Unit 1 Floating-point Unit 2	Floating Point Operations	3

Example Execution

Sequential Execution

```
CYCLE  1 xseed1 = xseed * 1309
CYCLE  2 nop
CYCLE  3 nop
CYCLE  4 yseed1 = yseed * 1308
CYCLE  5 nop
CYCLE  6 nop
CYCLE  7 xseed2 = xseed1 + 13849
CYCLE  8 yseed2 = yseed1 + 13849
CYCLE  9 xseed  = xseed2 && 65535
CYCLE  0 yseed  = yseed2 && 65535
CYCLE 11 tseed1 = tseed * 1307
CYCLE 12 nop
CYCLE 13 nop
CYCLE 14 vseed1 = vseed * 1306
CYCLE 15 nop
CYCLE 16 nop
CYCLE 17 tseed2 = tseed1 + 13849
CYCLE 18 vseed2 = vseed1 + 13849
CYCLE 19 tseed  = tseed2 && 65535
CYCLE 20 vseed  = vseed2 && 65535
CYCLE 21 xsq = xseed * xseed
CYCLE 22 nop
CYCLE 23 nop
CYCLE 24 ysq = yseed * yseed
CYCLE 25 nop
CYCLE 26 nop
CYCLE 27 xysumsq = xsq + ysq
CYCLE 28 tsq = tseed * tseed
CYCLE 29 nop
CYCLE 30 nop
CYCLE 31 vsq = vseed * vseed
CYCLE 32 nop
CYCLE 33 nop
CYCLE 34 tvsumsq = tsq + vsq
CYCLE 35 plc = plc + 1
CYCLE 36 tp = tp + 2
CYCLE 37 if xysumsq > radius goto @xy-no-hit
```

ILP Execution

```
               INT ALU            INT ALU            FLOAT ALU           FLOAT ALU
CYCLE  1 tp=tp+2              plc=plc+1          vseed1=vseed*1306   tseed1=tseed*1307
CYCLE  2                                         yseed1=yseed*1308   xseed1=xseed*1309
CYCLE  3 nop
CYCLE  4 vseed2=vseed1+13849  tseed2=tseed1+13849
CYCLE  5 yseed2=yseed1+13849  xseed2=xseed1+13849
CYCLE  6 yseed=yseed2&&65535  xseed=xseed2&&65535
CYCLE  7 vseed=vseed2&&65535  tseed=tseed2&&65535 ysq=yseed*yseed    xsq=xseed*xseed
CYCLE  8                                         vsq=vseed*vseed     tsq=tseed*tseed
CYCLE  9 nop
CYCLE  0 xysumsq=xsq+ysq
CYCLE 11 tvsumsq=tsq+vsq        if xysumsq>radius goto @xy-no-hit
```

ILP: Instruction-Level Parallelism

- **ILP is is a measure of the amount of inter-dependencies between instructions**
- **Average ILP = no. instruction / no. cyc required**

code1: ILP = 1

i.e. must execute serially

code2: ILP = 3

i.e. can execute at the same time

code1:	r1 ← r2 + 1 r3 ← r1 / 17 r4 ← r0 - r3

code2:	r1 ← r2 + 1 r3 ← r9 / 17 r4 ← r0 - r10

Inter-instruction Dependences

- *Data dependence*

  $r_3 \leftarrow r_1$ op r_2 Read-after-Write
  $r_5 \leftarrow r_3$ op r_4 (RAW)

- *Anti-dependence*

  $r_3 \leftarrow r_1$ op r_2 Write-after-Read
  $r_1 \leftarrow r_4$ op r_5 (WAR)

- *Output dependence*

  $r_3 \leftarrow r_1$ op r_2 Write-after-Write
  $r_5 \leftarrow r_3$ op r_4 (WAW)
  $r_3 \leftarrow r_6$ op r_7

- Control dependence

Scope of ILP Analysis

ILP=1
r1 ⇐ r2 + 1
r3 ⇐ r1 / 17
r4 ⇐ r0 - r3

r11 ⇐ r12 + 1
r13 ⇐ r19 / 17
r14 ⇐ r0 - r20

ILP=2 (all six instructions)

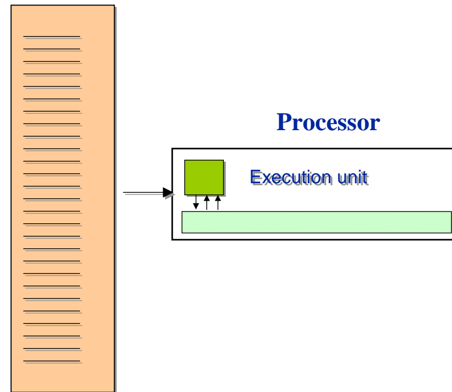
Out-of-order execution permits more ILP to be exploited

Questions Facing ILP System Designers

- **What gives rise to instruction-level parallelism in conventional, sequential programs**
- **How is the potential parallelism identified and enhanced, and how much is there?**
- **What must be done in order to exploit the parallelism that has been identified?**
- **How should the work of identifying, enhancing and exploiting the parallelism be divided between the hardware and software (the compiler)?**
- **What are the alternatives in selecting the architecture of an ILP processor?**

Sequential Processor

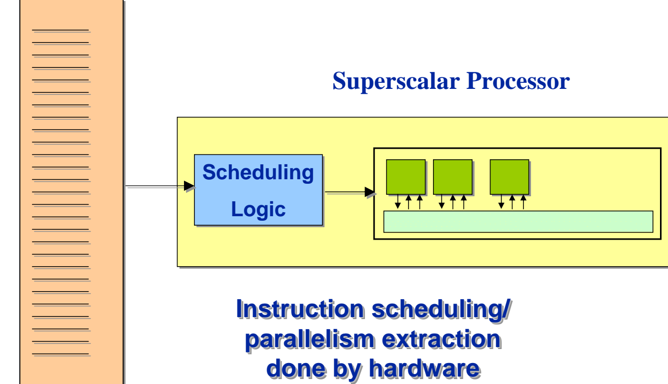
Sequential Instructions



CS211 17

ILP Processors:Superscalar

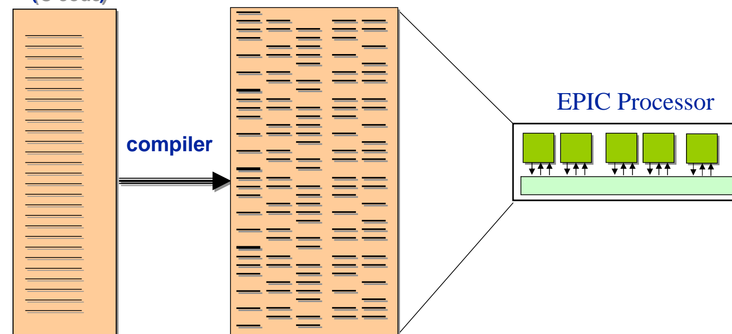
Sequential Instructions



Instruction scheduling/ parallelism extraction done by hardware

CS211 18

ILP Processors:EPIC/VLIW



CS211 19

ILP Architectures

- **Between the compiler and the run-time hardware, the following functions must be performed**
  - Dependencies between operations must be determined
  - Operations that are independent of any operation that has not yet completed must be determined
  - Independent operations must be scheduled to execute at some particular time, on some specific functional unit, and must be assigned a register into which the result may be deposited.

CS211 20

ILP Architecture Classifications

- Sequential Architectures
  - The program is not expected to convey any explicit information regarding parallelism
- Dependence Architectures
  - The program explicitly indicates dependencies between operations
- Independence Architectures
  - The program provides information as to which operations are independent of one another

Sequential Architecture

- Program contains no explicit information regarding dependencies that exist between instructions
- Dependencies between instructions must be determined by the hardware
  - It is only necessary to determine dependencies with sequentially preceding instructions that have been issued but not yet completed
- Compiler may re-order instructions to facilitate the hardware's task of extracting parallelism

Sequential Architecture Example

- Superscalar processor is a representative ILP implementation of a sequential architecture
  - For every instruction issued by a Superscalar processor, the hardware must check whether the operands interfere with the operands of any other instruction that is either
    - » (1) already in execution, (2) been issued but waiting for completion of interfering instructions that would have been executed earlier in a sequential program, and (3) being issued concurrently but would have been executed earlier in the sequential execution of the program
  - Superscalar proc. issues multiple inst. In cycle

Sequential Architecture Example

- Superscalar processors attempt to issue multiple instructions per cycle
  - However, essential dependencies are specified by sequential ordering so operations must be processed in sequential order
  - This proves to be a performance bottleneck that is very expensive to overcome
- Alternative to multiple instructions per cycle is pipelining and issue instructions faster

Dependence Architecture

- Compiler or programmer communicates to the hardware the dependencies between instructions
  - removes the need to scan the program in sequential order (the bottleneck for superscalar processors)
- Hardware determines at run-time when to schedule the instruction

Dependence Architecture Example

- Dataflow processors are representative of Dependence architectures
  - Execute instruction at earliest possible time subject to availability of input operands and functional units
  - Dependencies communicated by providing with each instruction a list of all successor instructions
  - As soon as all input operands of an instruction are available, the hardware fetches the instruction
  - The instruction is executed as soon as a functional unit is available
- Few Dataflow processors currently exist

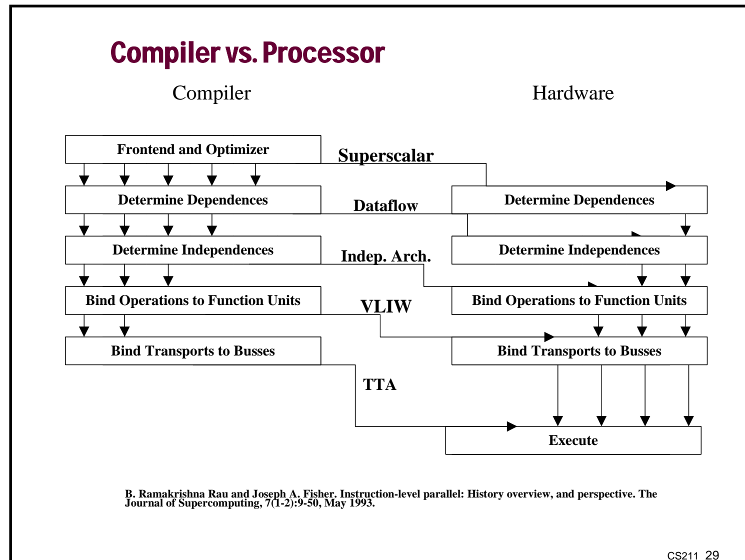
Independence Architecture

- By knowing which operations are independent, the hardware needs no further checking to determine which instructions can be issued in the same cycle
- The set of independent operations is far greater than the set of dependent operations
  - Only a subset of independent operations are specified
- The compiler may additionally specify on which functional unit and in which cycle an operation is executed
  - The hardware needs to make no run-time decisions

Independence Architecture Example

- EPIC/VLIW processors are examples of Independence architectures
  - Specify exactly which functional unit each operation is executed on and when each operation is issued
  - Operations are independent of other operations issued at the same time as well as those that are in execution
  - Compiler emulates at compile time what a dataflow processor does at run-time

Compiler vs. Processor



B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel: History overview, and perspective. The Journal of Supercomputing, 7(1-2):9-50, May 1993.

VLIW and Superscalar

- basic structure of VLIW and superscalar consists of a number of Eus, each capable of parallel operation on data fetched from register file
- VLIW and superscalar require highly multiported register files
  - limit on register ports places inherent limitation on maximum number of EUs

VLIW & Superscalar-Differences

- presentation of instructions:
  - VLIW receive multi-operation instructions
  - Superscalar accept traditional sequential stream but can issue more than one instruction
- VLIW needs very long instructions in order to specify what each EU should do
- Superscalar receive stream of conventional instructions

VLIW&Superscalar-Differences

- Decode and Issue unit in superscalar issues multiple instructions for the EUs
  - Have to figure out dependencies and independent instructions
- VLIW expect dependency free code whereas superscalar typically do not expect this.
  - Superscalars cope with dependencies using hardware

Instruction Scheduling

- dependencies must be detected and resolved
- *static:* accomplished by compiler which avoids dependencies by rearranging code
- *dynamic:* detection and resolution performed by hardware. processor typically maintains issue window (prefetched inst) and execution window (being executed). check for dependencies in issue window.

Instruction Scheduling: The Optimization Goal

- *Given a source program P, schedule the instructions so as to minimize the overall execution time on the functional units in the target machine.*

EPIC/VLIW vs Superscalar: Summary

- In EPIC processors
  - compiler manages hardware resources
  - synergy between compiler and architecture is key
  - some compiler optimizations will be covered in depth
- In Superscalar processors
  - architecture is “self-managed”
  - notably instruction dependence analysis and scheduling done by hardware

Next. . . .

- Basic ILP techniques: dependence analysis, simple code optimization
  - First look at S/W (Compiler) technique to extract ILP
- Superscalar Processors/ Dynamic ILP
  - Branches
  - scheduling algorithms implemented in hardware
- EPIC Processors
  - Intel IA64 family
  - compiler optimizations needed
- Overview of Compiler Optimization

Superscalar Processors

Superscalar Terminology

•Basic

Superscalar	Able to issue > 1 instruction / cycle
Superpipelined	Deep, but not superscalar pipeline. E.g., MIPS R5000 has 8 stages
Branch prediction	Logic to guess whether or not branch will be taken, and possibly branch target

•Advanced

Out-of-order	Able to issue instructions out of program order
Speculation	Execute instructions beyond branch points, possibly nullifying later
Register renaming	Able to dynamically assign physical registers to instructions
Retire unit	Logic to keep track of instructions as they complete.

Superscalar Execution Example

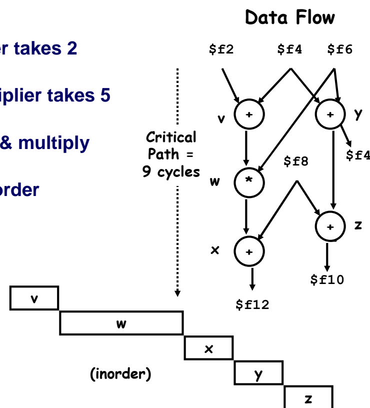
Single Order, Data Dependence – In Order

- Assumptions
  - Single FP adder takes 2 cycles
  - Single FP multiplier takes 5 cycles
  - Can issue add & multiply together
  - Must issue in-order
  - <op> in,in,out

(In order)

(Single adder, data dependence)

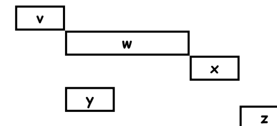
```
v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
```



Adding Advanced Features

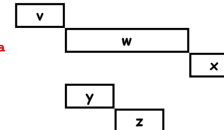
- Out Of Order Issue
  - Can start y as soon as adder available
  - Must hold back z until `$f10` not busy & adder available

```
v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
```

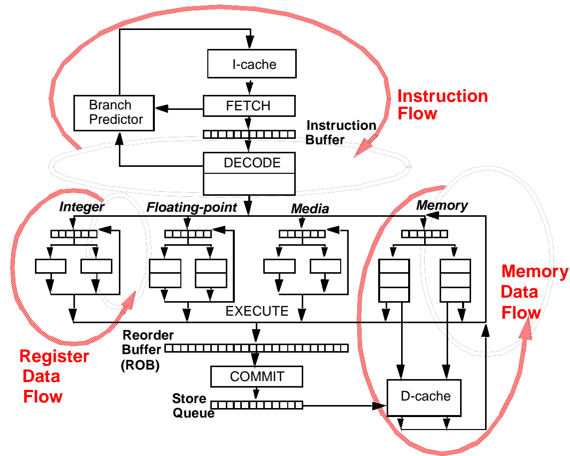


- With Register Renaming

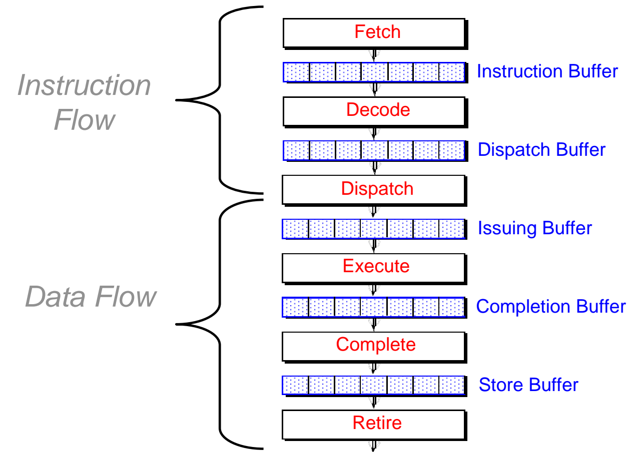
```
v:  addt  $f2, $f4, $f10a
w:  mult  $f10a, $f6, $f10a
x:  addt  $f10a, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f14
```



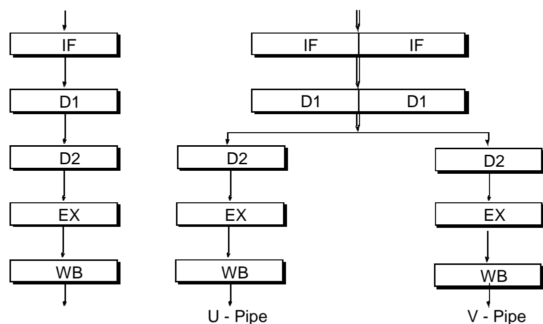
Flow Path Model of Superscalars



Superscalar Pipeline Design



Inorder Pipelines

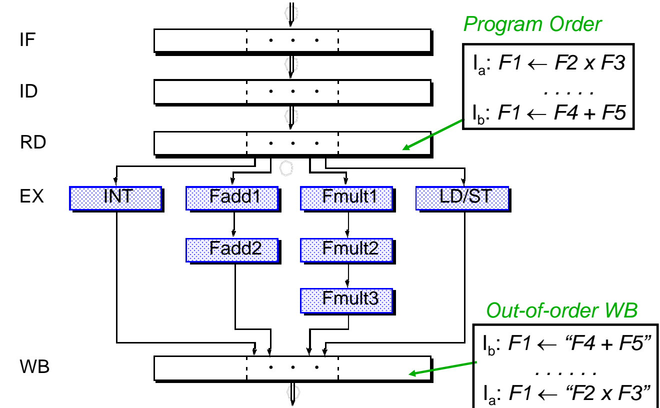


Intel i486

Intel Pentium

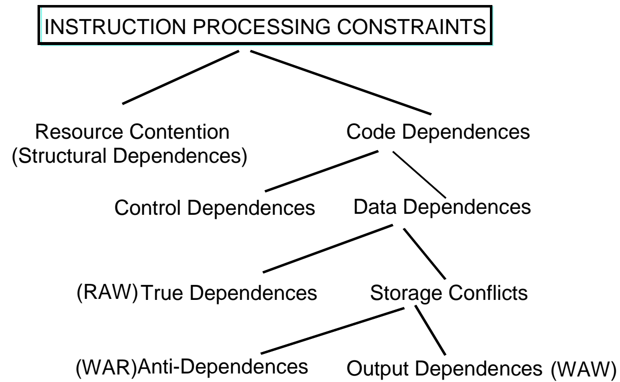
Inorder pipeline, no WAW no WAR (almost always true)

Out-of-order Pipelining 101



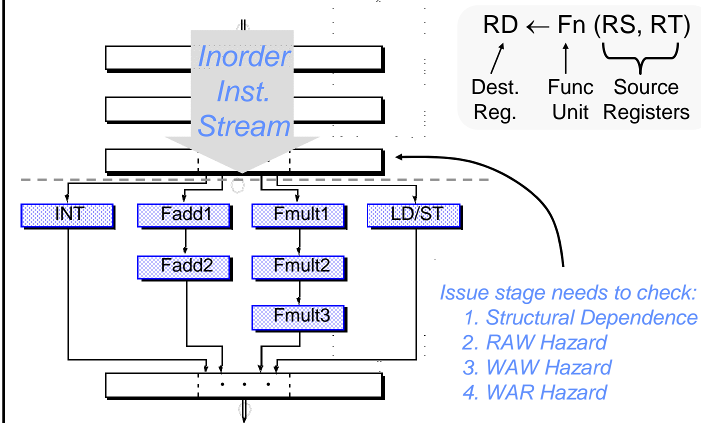
What is the value of F1? WAW!!!

Superscalar Execution Check List



CS211 45

In-order Issue into Diversified Pipelines



CS211 46

Superscalar Processors

- **Tasks:**
- parallel decoding
- superscalar instruction issue
- parallel instruction execution
  - ***preserving sequential consistency of exception processing***
  - ***preserving sequential consistency of exec.***

CS211 47

Parallel Execution

- **when instructions executed in parallel they will finish *out of program order***
  - **unequal execution times**
- **specific means needed to preserve logical consistency**
  - preservation of sequential consistency
- ***exceptions* during execution**
  - **preservation seq. consistency exception proc.**

CS211 48

More Hardware Features to Support ILP

- Pipelining
  - Advantages
    - » Relatively low cost of implementation - requires latches within functional units
    - » With pipelining, ILP can be doubled, tripled or more
  - Disadvantages
    - » Adds delays to execution of individual operations
    - » Increased latency eventually counterbalances increase in ILP

Hardware Features to Support ILP

- Additional Functional Units
  - Advantages
    - » Does not suffer from increased latency bottleneck
  - Disadvantages
    - » Amount of functional unit hardware proportional to degree of parallelism
    - » Interconnection network and register file size proportional to square of number of functional units

Hardware Features to Support ILP

- Instruction Issue Unit
  - Care must be taken not to issue an instruction if another instruction upon which it is dependent is not complete
  - Requires complex control logic in Superscalar processors
  - Virtually trivial control logic in VLIW processors

Hardware Features to Support ILP

- Speculative Execution
  - Little ILP typically found in basic blocks
    - » a straight-line sequence of operations with no intervening control flow
  - Multiple basic blocks must be executed in parallel
    - » Execution may continue along multiple paths before it is known which path will be executed

Hardware Features to Support ILP

- Requirements for Speculative Execution
  - Terminate unnecessary speculative computation once the branch has been resolved
  - Undo the effects of the speculatively executed operations that should not have been executed
  - Ensure that no exceptions are reported until it is known that the excepting operation should have been executed
  - Preserve enough execution state at each speculative branch point to enable execution to resume down the correct path if the speculative execution happened to proceed down the wrong one.

Hardware Features to Support ILP

- Speculative Execution
  - Expensive in hardware
  - Alternative is to perform speculative code motion at compile time
    - » Move operations from subsequent blocks up past branch operations into proceeding blocks
  - Requires less demanding hardware
    - » A mechanism to ensure that exceptions caused by speculatively scheduled operations are reported if and only if flow of control is such that they would have been executed in the non-speculative version of the code
    - » Additional registers to hold the speculative execution state

Introduction to S/W Techniques for ILP

Instruction Level Parallelism (ILP)

- ILP: Overlap execution of unrelated instructions
- How to extract parallelism from program?
  - Beyond single block to get more instruction level parallelism
- Who does instruction scheduling and parallelism extraction?
  - Software or Hardware or mix ?
  - Superscalar processors require H/W solutions, but can also use some compiler help
- What new hardware features are required to support more ILP..?
  - Different requirements for Superscalar and EPIC

ILP Techniques

- Key issue to worry about is Hazards
  - Control and data
  - Rising out of dependencies
  - Introduces stalls in execution
- How to increase ILP
  - Reduce data hazards: RAW, WAR, WAW
  - Handle control hazards better
  - Increase ideal IPC (instructions per cycle)
- Note: Bottom line is how to better schedule instructions

Recall our old friend from Review of pipelining

- CPI = ideal CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal (pipeline) CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Ideas to Reduce Stalls

	Technique	Reduces
Chapter 3	Dynamic scheduling	Data hazard stalls
Chapter 3	Dynamic branch prediction	Control stalls
Chapter 3	Issuing multiple instructions per cycle	Ideal CPI
Chapter 3	Speculation	Data and control stalls
Chapter 3	Dynamic memory disambiguation	Data hazard stalls involving memory
Chapter 4	Loop unrolling	Control hazard stalls
Chapter 4	Basic compiler pipeline scheduling	Data hazard stalls
Chapter 4	Compiler dependence analysis	Ideal CPI and data hazard stalls
Chapter 4	Software pipelining and trace scheduling	Ideal CPI and data hazard stalls
Chapter 4	Compiler speculation	Ideal CPI, data and control stalls

Instruction-Level Parallelism (ILP)

- Basic Block (BB) ILP is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25% => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks
- Simplest: loop-level parallelism to exploit parallelism among iterations of a loop
  - Vector is one way
    - » Where is this useful ?
  - If not vector, then either dynamic via branch prediction or static via loop unrolling by compiler

Quick recall of data hazards..

- True/flow dependencies - RAW
- Name dependencies WAR, WAW
  - Also known as false dependencies, output dep

Data Dependence and Hazards

- $Instr_J$ is data dependent on $Instr_I$
  $Instr_J$ tries to read operand before $Instr_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- or $Instr_J$ is data dependent on $Instr_K$ which is dependent on $Instr_I$
- Caused by a "True Dependence" (compiler term)
- If true dependence caused a hazard in the pipeline, called a Read After Write (RAW) hazard

Name Dependence #1: Anti-dependence

- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
- $Instr_J$ writes operand *before* $Instr_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

  Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard

Name Dependence #2: Output dependence

- $Instr_J$ writes operand *before* $Instr_I$ writes it.

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard

ILP and Data Hazards

- HW/SW must preserve program order:
  - order instructions would execute in if executed sequentially one at a time as determined by original source program
  - Does this mean we can never change order of execution of instructions ?
    - » Ask - What happens if we change the order of an instruction
    - » Does result change ?

ILP and Data Hazards

- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict
  - Register renaming resolves name dependence for regs
  - Either by compiler or by HW
-

Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```
if p1 {
  S1;
};
if p2 {
  S2;
}
```

- `S1` is control dependent on `p1`, and `S2` is control dependent on `p2` but not on `p1`.

Control Dependence Ignored

- Control dependence need not be preserved
  - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program
- Instead, 2 properties critical to program correctness are exception behavior and data flow

Exception Behavior

- Preserving exception behavior => any changes in instruction execution order must not change how exceptions are raised in program (=> no new exceptions)
- Example:

```
    DADDU     R2,R3,R4
    BEQZ      R2,L1
    LW        R1,0(R2)
L1:
```

- Problem with moving `LW` before `BEQZ`?

Data Flow

- Data flow: actual flow of data values among instructions that produce results and those that consume them
  - branches make flow dynamic, determine which instruction is supplier of data
- Example:

```
DADDU    R1,R2,R3
BEQZ     R4,L
DSUBU    R1,R5,R6
L:  …
OR       R7,R1,R8
```

- `OR` depends on `DADDU` or `DSUBU`? Must preserve data flow on execution

Ok.. ILP through Software/Compiler

- Ask what you (SW/compiler) can do for the HW ?
- Quick look at one SW technique to
  - Decrease CPU time
  - expose more ILP

Loop Unrolling: A Simple S/W Technique

- Parallelism within one "basic block" is minimal
  - Need to look at larger regions of code to schedule
- Loops are very common
  - Number of iterations, same tasks in each iteration
- Simple Observation : If iterations are independent, then multiple iterations can be executed in parallel
- Loop Unrolling- Unrolling multiple iterations of a loop to create more instructions to schedule

Example FP Loop: Where are the Hazards?

```
Loop: LD    F0,0(R1)   ;F0=vector element
      ADDD  F4,F0,F2   ;add scalar from F2
      SD    0(R1),F4   ;store result
      SUBI  R1,R1,8    ;decrement pointer 8B (DW)
      BNEZ  R1,Loop    ;branch R1!=zero
      NOP              ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- Where are the stalls?

FP Loop Hazards

```
Loop: LD    F0,0(R1)   ;F0=vector element
      ADDD  F4,F0,F2   ;add scalar in F2
      SD    0(R1),F4   ;store result
      SUBI  R1,R1,8    ;decrement pointer 8B (DW)
      BNEZ  R1,Loop    ;branch R1!=zero
      NOP              ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

FP Loop Showing Stalls

```
1 Loop: LD    F0,0(R1)   ;F0=vector element
2       stall
3       ADDD  F4,F0,F2   ;add scalar in F2
4       stall
5       stall
6       SD    0(R1),F4   ;store result
7       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
8       BNEZ  R1,Loop    ;branch R1!=zero
9       stall            ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: LD    F0,0(R1)
2       stall
3       ADDD  F4,F0,F2
4       SUBI  R1,R1,8
5       BNEZ  R1,Loop    ;delayed branch
6       SD    8(R1),F4   ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks: Unroll loop 4 times code to make faster?

Unroll Loop Four Times (straightforward way)

```
1  Loop: LD    F0,0(R1)
2        ADDD  F4,F0,F2
3        SD    0(R1),F4      ;drop SUBI & BNEZ
4        LD    F6,-8(R1)
5        ADDD  F8,F6,F2
6        SD    -8(R1),F8     ;drop SUBI & BNEZ
7        LD    F10,-16(R1)
8        ADDD  F12,F10,F2
9        SD    -16(R1),F12   ;drop SUBI & BNEZ
10       LD    F14,-24(R1)
11       ADDD  F16,F14,F2
12       SD    -24(R1),F16
13       SUBI  R1,R1,#32     ;alter to 4*8
14       BNEZ  R1,LOOP
15       NOP
```

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4

Unrolled Loop That Minimizes Stalls

```
1  Loop: LD    F0,0(R1)
2        LD    F6,-8(R1)
3        LD    F10,-16(R1)
4        LD    F14,-24(R1)
5        ADDD  F4,F0,F2
6        ADDD  F8,F6,F2
7        ADDD  F12,F10,F2
8        ADDD  F16,F14,F2
9        SD    0(R1),F4
10       SD    -8(R1),F8
11       SD    -16(R1),F12
12       SUBI  R1,R1,#32
13       BNEZ  R1,LOOP
14       SD    8(R1),F16     ; 8-32 = -24
```

- What assumptions made when moved code?
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration
When safe to move instructions?

Compiler Perspectives on Code Movement

- Definitions: compiler concerned about dependencies in program, whether or not a HW hazard depends on a given pipeline
- Try to schedule to avoid hazards
- (True) Data dependencies (RAW if a hazard for HW)
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- If dependent, can't execute in parallel
- Easy to determine for registers (fixed names)
- Hard for memory:
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

Where are the data dependencies?

```
1  Loop: LD    F0,0(R1)
2        ADDD  F4,F0,F2
3        SUBI  R1,R1,8
4        BNEZ  R1,Loop   ;delayed branch
5        SD    8(R1),F4  ;altered when move past SUBI
```

Where are the name dependencies?

```
1  Loop: LD     F0,0(R1)
2        ADDD   F4,F0,F2
3        SD     0(R1),F4      ;drop SUBI & BNEZ
4        LD     F0,-8(R1)
2        ADDD   F4,F0,F2
3        SD     -8(R1),F4     ;drop SUBI & BNEZ
7        LD     F0,-16(R1)
8        ADDD   F4,F0,F2
9        SD     -16(R1),F4    ;drop SUBI & BNEZ
10       LD     F0,-24(R1)
11       ADDD   F4,F0,F2
12       SD     -24(R1),F4
13       SUBI   R1,R1,#32     ;alter to 4*8
14       BNEZ   R1,LOOP
15       NOP
```

How can remove them?

CS211 81

Where are the name dependencies?

```
1  Loop: LD     F0,0(R1)
2        ADDD   F4,F0,F2
3        SD     0(R1),F4      ;drop SUBI & BNEZ
4        LD     F6,-8(R1)
5        ADDD   F8,F6,F2
6        SD     -8(R1),F8     ;drop SUBI & BNEZ
7        LD     F10,-16(R1)
8        ADDD   F12,F10,F2
9        SD     -16(R1),F12   ;drop SUBI & BNEZ
10       LD     F14,-24(R1)
11       ADDD   F16,F14,F2
12       SD     -24(R1),F16
13       SUBI   R1,R1,#32     ;alter to 4*8
14       BNEZ   R1,LOOP
15       NOP
```

Called “register renaming”

CS211 82

Compiler Perspectives on Code Movement

- Again Name Dependenceis are Hard for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Our example required compiler to know that if R1 doesn't change then:

$0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$

There were no dependencies between some loads and stores so they could be moved by each other

CS211 83

Compiler Perspectives on Code Movement

- Final kind of dependence called control dependence
- Example

```
if p1 {S1;};
if p2 {S2;};
```

S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.

CS211 84

Compiler Perspectives on Code Movement

- Another kind of dependence called name dependence: two instructions use same name (register or memory location) but don't exchange data
- Antidependence (WAR if a hazard for HW)
  - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
- Output dependence (WAW if a hazard for HW)
  - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

Compiler Perspectives on Code Movement

- Two (obvious) constraints on control dependences:
  - An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
  - An instruction that is not control dependent on a branch cannot be moved to after the branch so that its execution is controlled by the branch.
- Control dependencies relaxed to get parallelism; get same effect if preserve order of exceptions (address in register checked by branch before use) and data flow (value in register depends on branch)
  - Can "violate" the two constraints above by placing some 'checks' in place ?
    - » Branch prediction, speculation

Where are the control dependencies?

```
1  Loop: LD    F0,0(R1)
2        ADDD  F4,F0,F2
3        SD    0(R1),F4
4        SUBI  R1,R1,8
5        BEQZ  R1,exit
6        LD    F0,0(R1)
7        ADDD  F4,F0,F2
8        SD    0(R1),F4
9        SUBI  R1,R1,8
10       BEQZ  R1,exit
11       LD    F0,0(R1)
12       ADDD  F4,F0,F2
13       SD    0(R1),F4
14       SUBI  R1,R1,8
15       BEQZ  R1,exit
....
```


When Safe to Unroll Loop?

- Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)

```
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i];    /* S1 */
    B[i+1] = B[i] + A[i+1];} /* S2 */
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.

2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
This is a "loop-carried dependence": between iterations

- Implies that iterations are dependent, and can't be executed in parallel
- Not the case for our prior example; each iteration was distinct

HW Schemes: Instruction Parallelism

- Why in HW at run time?
  - Works when can't know real dependence at compile time
  - Compiler simpler
  - Code for one machine runs well on another
- Key idea: Allow instructions behind stall to proceed

```
DIVD  F0,F2,F4
ADDD  F10,F0,F8
SUBD  F12,F8,F14
```

  - Enables out-of-order execution => out-of-order completion
  - ID stage checked both for structuralScoreboard dates to CDC 6600 in 1963

Next. . . Superscalar Processor Design

- How to deal with instruction flow
  - Dynamic Branch prediction
- How to deal with register/data flow
  - Register renaming
- Dynamic branch prediction
- Dynamic scheduling using Tomasulo method