



CS 211
**Introduction to Explicitly Parallel
Instruction Computing (EPIC)
Architectures**

Bhagi Narahari



EPIC Architectures Overview

- History
- Overview of key EPIC concepts
 - speculation, predication, register files
- Introduction to Compiler Optimization

CS 211



The EPIC Concept- A little history

- The great CISC vs RISC debate
 - who won?
- Intel's "combination approach"
 - CISC instructions running on RISC core (486)
 - adding superscalar features (Pentium)
 - RISC proc had CISC like features for special units
- 64 bit RISC instructions
 - HP PA-RISC project, Early Intel efforts
 - iterative improvements but no revolution
- To make programs run faster need to look at new architectures
 - something radical had to be done for more performance

CS 211



VLIW and EPIC

- VLIW architectures progressed to EPIC
- A quick look at "pure" VLIW approach

CS 211

What Is VLIW?

- VLIW hardware is simple and straightforward, like SIMD machines.
- While SIMD broadcasts one instruction, VLIW separately directs each functional unit.

SIMD Instruction Execution

VLIW Instruction Execution

CS 211

Historical Perspective: ~~Microcoding, nanocoding (and RISC)~~

Macro Instructions → micro sequencer → microcode store → datapath control

nanocode store → datapath control

CS 211

Horizontal Microcode and VLIW

- A generation of high-performance, application-specific computers relied on *horizontally* microprogrammed computing engines.

Microsequencer (2910) → Microcode Memory → Bit Slice ALU

- Aggressive (but tedious, hand programming) at the microcode level provided performance well above sequential processors.

CS 211

Principles of VLIW Operation

- Statically scheduled ILP architecture.
- Wide instructions specify many independent simple operations.

VLIW Instruction (100 - 1000 bits)

- Multiple functional units executes all of the operations in an instruction concurrently, providing fine-grain parallelism within each instruction
- Instructions directly control the hardware with no interpretation and minimal decoding.
- A powerful optimizing compiler is responsible for locating and extracting ILP from the program and for scheduling operations to exploit the available parallel resources.

CS 211



Formal VLIW Models

- Josh Fisher proposed the first VLIW machine at Yale (1983)
- Fisher's *Trace Scheduling* algorithm for microcode compaction could exploit more ILP than any existing processor could provide.
- The ELI-512 was to provide massive resources to a single instruction stream
 - 16 processing clusters- multiple functional units/cluster.
 - partial crossbar interconnect.
 - multiple memory banks.
 - attached processor – no I/O, no operating system.
- Later VLIW models became increasingly more regular
 - Compiler complexity was a greater issue than originally envisioned

CS 211



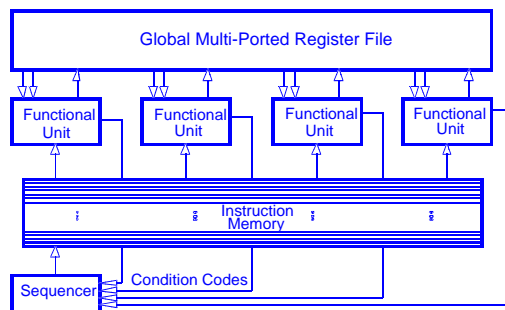
Ideal Models for VLIW Machines

- Almost all VLIW research has been based upon an ideal processor model.
- This is primarily motivated by compiler algorithm developers to simplify scheduling algorithms and compiler data structures.
 - This model includes:
 - Multiple universal functional units
 - Single-cycle global register file
 - and often:
 - Single-cycle execution
 - Unrestricted, Multi-ported memory
 - Multi-way branching
 - and sometimes:
 - Unlimited resources (Functional units, registers, etc.)

CS 211



VLIW Execution Characteristics



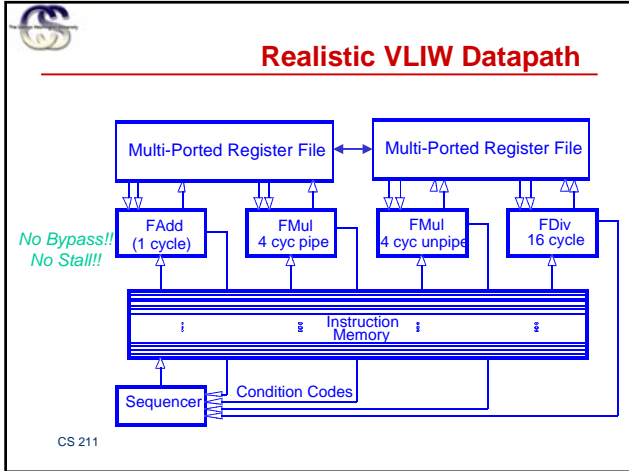
CS 211 Basic VLIW architectures are a generalized form of horizontally microprogrammed machines



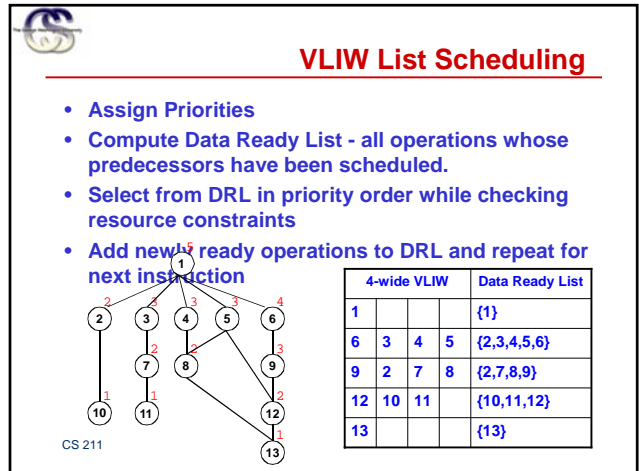
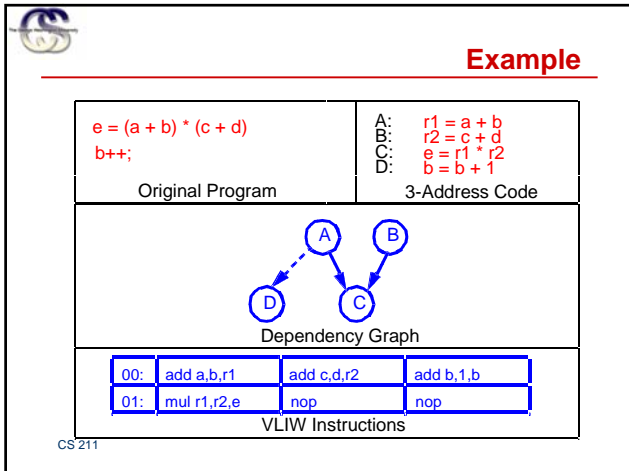
VLIW Design Issues

- Unresolved design issues
 - The best functional unit mix
 - Register file and interconnect topology
 - Memory system design
 - Best instruction format
- Many questions could be answered through experimental research
 - Difficult - needs effective retargetable compilers
- Compatibility issues still limit interest in general-purpose VLIW technology

CS 211 However, VLIW may be the only way to build 8-16 operation/cycle machines.



- ### Scheduling for Fine-Grain Parallelism
- The program is translated into primitive RISC-style (three address) operations
 - Dataflow analysis is used to derive an operation precedence graph from a portion of the original program
 - Operations which are independent can be scheduled to execute concurrently contingent upon the availability of resources
 - The compiler manipulates the precedence graph through a variety of semantic-preserving transformations to expose additional parallelism
- CS 211





Enabling Technologies for VLIW

- VLIW Architectures achieve high performance through the combination of a number of key enabling *hardware* and *software* technologies.
 - Optimizing Schedulers (compilers)
 - Static Branch Prediction
 - Symbolic Memory Disambiguation
 - Predicated Execution
 - (Software) Speculative Execution
 - Program Compression

CS 211



Strengths of VLIW Technology

- Parallelism can be exploited at the instruction level
 - Available in both vectorizable and sequential programs.
- Hardware is regular and straightforward
 - Most hardware is in the datapath performing useful computations.
 - Instruction issue costs scale approximately linearly
Potentially very high clock rate
- Architecture is “*Compiler Friendly*”
 - Implementation is completely exposed - 0 layer of interpretation
 - Compile time information is easily propagated to run time.
- Exceptions and interrupts are easily managed
- Run-time behavior is highly predictable
 - Allows real-time applications.
 - Greater potential for code optimization.

CS 211



Weaknesses of VLIW Technology

- No object code compatibility between generations
- Program size is large (explicit NOPs)
 - Multiflow machines predated “dynamic memory compression” by encoding NOPs in the instruction memory
- Compilers are extremely complex
 - Assembly code is almost impossible
- Philosophically incompatible with caching techniques
- VLIW memory systems can be very complex
 - Simple memory systems may provide very low performance
 - Program controlled multi-layer, multi-banked memory
- Parallelism is underutilized for some algorithms.

CS 211



VLIW vs. Superscalar [Bob Rau, HP]

Attributes	Superscalar	VLIW
Multiple instructions/cycle	yes	yes
Multiple operations/instruction stream	no	yes
Runtime analysis of register dependencies	yes	no
Run-time analysis of memory dependencies	maybe	occasionally
Runtime instruction reordering	maybe (Resv. stations)	no
Runtime register allocation	maybe (renaming)	maybe (iteration frames)

CS 211

Real VLIW Machines

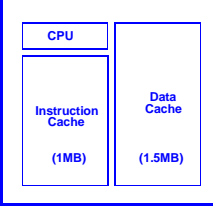
- **VLIW Minisupercomputers/Superminicomputers:**
 - Multiflow TRACE 7/300, 14/300, 28/300 [Josh Fisher]
 - Multiflow TRACE /500 [Bob Colwell]
 - Cydrome Cydra 5 [Bob Rau]
 - IBM Yorktown VLIW Computer (research machine)
- **Single-Chip VLIW Processors:**
 - Intel iWarp, Philip's LIFE Chips (research)
- **Single-Chip VLIW Media (through-put) Processors:**
 - Trimedia, Chromatic, Micro-Unity
- **DSP Processors (TI TMS320C6x)**

- Intel/HP EPIC IA-64 (Explicitly Parallel Instruction Comp.)
- Transmeta Crusoe (x86 on VLIW??)
- Sun MAJC (Microarchitecture for Java Computing)

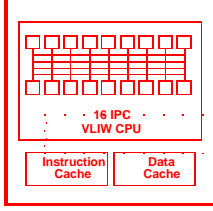
CS 211

Why VLIW Now?

1 Billion Transistor Superminicomputer



1 Billion Transistor VLIW Processor



- **Non-trivial ILP**
 - ILP and complexity
- **Better compilation technology**

CS 211

Performance Obstacles of Superscalars

- **Branches**
 - branch prediction helps, but penalty is still significant
 - limits scope of dynamic and static ILP analysis + code motion
- **Memory Load Latency**
 - CPU speed increases at 60% per year
 - memory speed increases only 5% per year
- **Memory Dependence**
 - disambiguation is hard, both in hardware and software
- **Sequential Execution Semantics ISAs**
 - total ordering of all the instructions
 - implicit inter-instruction dependences

Very expensive to implement wide dynamic superscalars

CS 211

Intel/HP EPIC/IA-64 Architecture

- **EPIC (Explicitly Parallel Instruction Computing)**
 - An ISA philosophy/approach
e.g. CISC, RISC, VLIW
 - Very closely related to but not the same as VLIW
- **IA-64**
 - An ISA definition
e.g. IA-32 (was called x86), PA-RISC
 - Intel's new 64-bit ISA
 - An EPIC type ISA
- **Itanium (was code named Merced)**
 - A processor implementation of an ISA
e.g. P6, PA8500
 - The first implementation of the IA-64 ISA

CS 211



IA-64 EPIC vs. Classic VLIW

- **Similarities:**
 - Compiler generated wide instructions
 - Static detection of dependencies
 - ILP encoded in the binary (a group)
 - Large number of architected registers
- **Differences:**
 - Instructions in a bundle can have dependencies
 - Hardware interlock between dependent instructions
 - Accommodates varying number of functional units and latencies
 - Allows dynamic scheduling and functional unit binding

Static scheduling are "suggestive" rather than absolute

 - ⇒ **Code compatibility across generations**

but software won't run at top speed until it is recompiled so "shrink-wrap binary" might need to include multiple builds

CS 211



EPIC Concepts

- **Explicitly Parallel Instruction Computing**
 - unlike early VLIW designs, EPIC does not use fixed width instructions....as many parallel as possible!
- **Programs must be written using sequential semantics**
 - parallel semantics not supported
 - explicitly lay out the parallelism
 - eg: swapping of operands

CS 211



EPIC: Key Concepts

- Speculation
- Predication (and parallel compares)
- Large (Rotating) Register Files

CS 211



EPIC - Philosophy

- **How do we increase performance**
 - work harder or
 - work smarter
- **To make the computer run faster**
 - Principle 1: make it do each thing faster
 - Principle 2: make it do more things at once
 - key to success of EPIC
- **neither running fast nor juggling lots of things at once is simple!**

CS 211



EPIC Concepts: Speculation

- What do you do with all the parallelism and how
 - traditional problem has been that we never have enough work ready in order to keep a machine fully busy
- what happens when you stop worrying about only doing things we must
 - if we have the power of parallelism, key is to not throw it away
 - anytime processor is ready to do six things, do not give it only two things to do and ignore ability to do more
 - how?

CS 211



EPIC Concepts: Speculation

- Speculatively ask machine to do more things
- pick tasks that might be needed in future
 - just aren't sure whether they will be needed at the time
 - make sure you can determine if they will be needed
 - extra tasks does not involve time (due to parallel units)
 - even if they useful only 50% of the time, we have completed 50% of the tasks ahead of time!
- Promise of EPIC based on speculation
 - goal is to compute things before they are needed, so when program needs result it is already there!

CS 211



EPIC Concepts: Predication

- Branching is generally bad because it interferes with the ideal pipeline model of reading instructions while earlier inst is executed
- ideally, if we eliminate branches then this problem disappears
- **Predication** is process by which branches are eliminated

CS 211



EPIC Concepts: Predication

- Predication allows instructions to execute in parallel, with some "on" and some "off" but without need for branches
 - Every instruction written with a specified predicate register to control whether instruction executes at run-time
- Ability to do "parallel compares"
 - ability to compute and combine comparison operations in parallel
 - (A>B) and (B<0) can be computed in parallel using parallel compare instructions

CS 211

EPIC Concepts: Predication

- EPIC provides predicated instructions
 - every instruction can be executed in predicated manner
 - instruction execution tied to result of a predicate register
 - one predicate register hardwired to a 1; use this to always execute

CS 211

EPIC Concepts: Predication

```

if (a > b) {
    x = a
    z = 1
} else {
    x = b
    z = z + 1
}
  
```

CS 211

EPIC Concepts: Predication

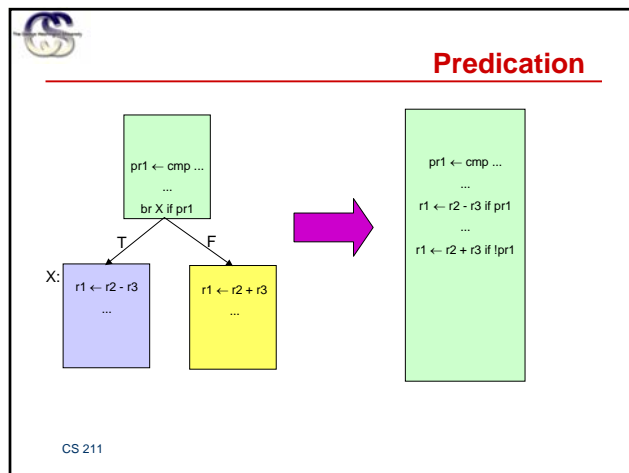
Test = TRUE if (a > b), else FALSE

```

if (test is TRUE) tmp1 = a
if (test is FALSE) tmp1 = b
x = tmp1
if (test is TRUE) tmp2 = 1
if (test is FALSE) tmp2 = z + 1
z = tmp2
  
```

note: No branches above!

CS 211



Predicated Execution

- Each instruction can be separately predicated
- 64 one-bit predicate registers in IA-64
An instruction is effectively a NOP if its predicate is false
- *Converts control flow into dataflow*

CS 211

Predication is not Control Speculation

- Two type of speculation:
 - data speculation
 - control speculation
- In **data** speculation: loads are moved ahead of stores (will be discussed later)
- In **control** speculation: instructions are moved from below a branch to above a branch
 - control speculation ≠ predication

CS 211

Differences

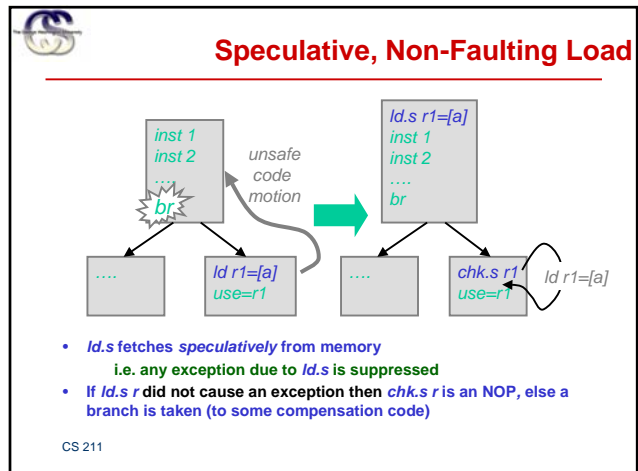
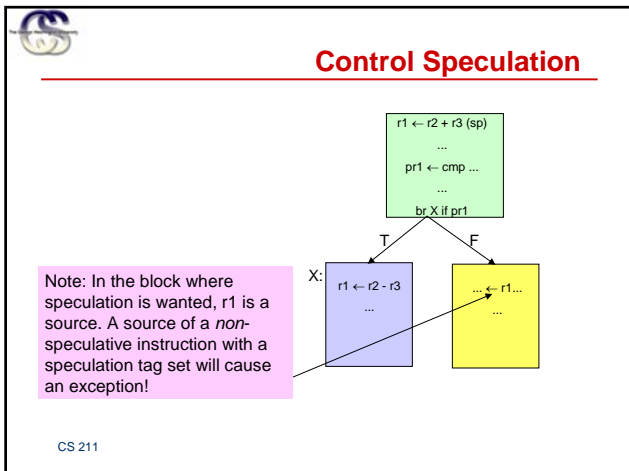
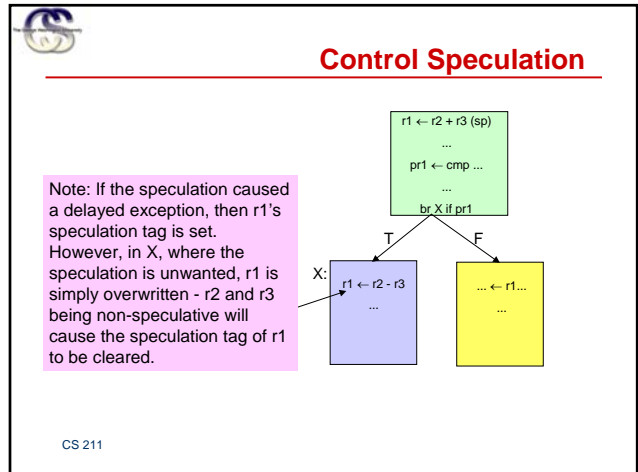
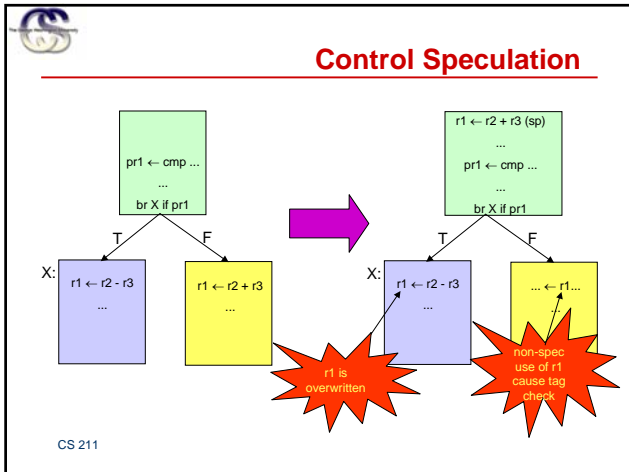
- In **predication**:
 - compare instruction sets predicate registers
 - predicate registers used to confirm instructions and annul others
 - exceptions take place where they occur

CS 211

Differences

- In **control speculation**:
 - instructions below a branch is moved to above it and marked as speculative
 - exceptions are postponed via register tagging
 - if speculation turns out to be false, result is discarded (register overwritten)
 - if speculation turns out to be true, must check whether speculative instructions caused exceptions

CS 211



Speculative, Non-Faulting Load

- Speculatively load data can be consumed prior to check
- “speculation” status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- *chk.s* checks the entire dataflow sequence for exceptions

CS 211

Speculative “Advanced” Load

- *ld.a* starts the monitoring of any store to the same address as the advanced load
- If no aliasing has occurred since *ld.a*, *ld.c* is a NOP
- If aliasing has occurred, *ld.c* re-loads from memory

CS 211

Using Speculative Load Results

CS 211

Compare Operations

- Comparison operations may
 - set predicate registers (up to two simultaneously)
 - compare to a GPR
- Comparison operations themselves can be predicated
- Predicate registers may be combined by logical operations

CS 211

If-conversion for Predication

- Identifying region of basic blocks based on resource requirement and profitability (branch mis-prediction rate, mis-prediction cost, and parallelism)
- Result: a predicated block

```

cmp.lt p1,p2=a,b;;
(p1) s = s + a;
(p2) s = s + b;;
*p = s
  
```

CS 211

Reducing Control Height with parallel compares

- Convert nested if's into a single predicate
- Result: shorter control path by reducing the number of branches

```

p1=0;;
cmp.lt.or p1,p0=a,b;
cmp.lt.or p1,p0=b,c;
(p1) br s1;;
s2
  
```

CS 211

Multiway Branch Example

- Use Multiway branches
 - Speculate compare (i.e. move above branch)
 - Do not reduce number of branches

```

cmp.lt p1,p0=a,b;
(p1) br X;;
cmp.lt p2,p0=b,c;
(p2) br Z;;
Y
  
```

```

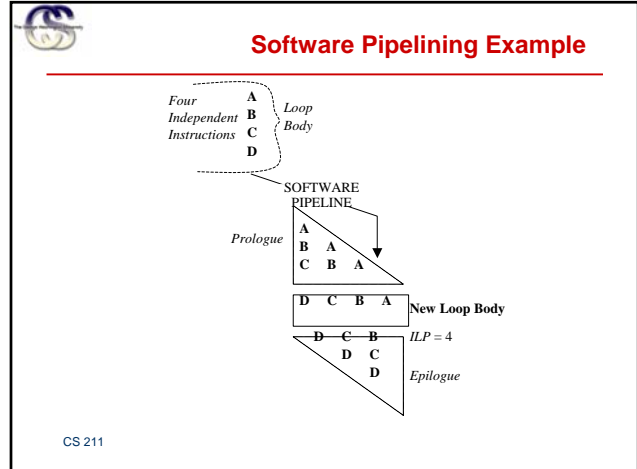
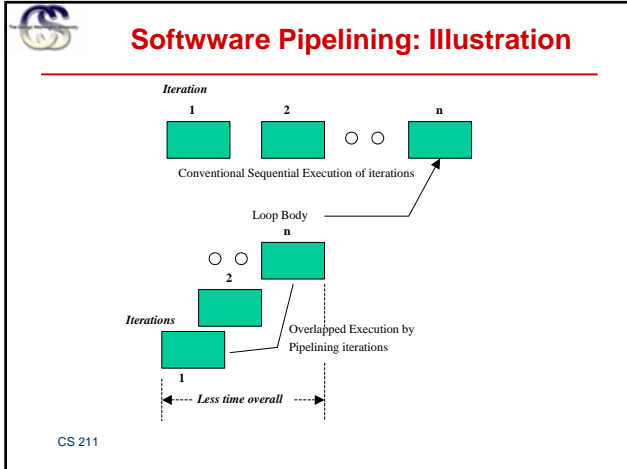
cmp.lt p1,p0=a,b;
cmp.lt p2,p0=b,c;
(p1) br X;
(p2) br Z;;
Y
  
```

CS 211

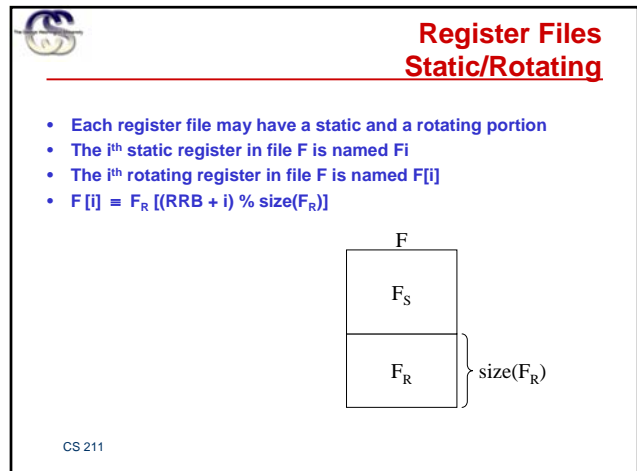
EPIC Concepts: Rotating Register Sets and Software Pipelining

- Speculation and Predication require large sets of registers in EPIC
- In addition, concept of Rotating Register Sets to support **Software Pipelining**
 - to help with execution of loops
 - extend pipeline concept

CS 211



- ### EPIC Concepts: Software Pipelining
- Software pipeline (also known as Modulo Scheduling) requires 'register reuse'
 - instead provide Rotating Register Sets
 - In IA-64 Registers organized into set of 32 static registers and 96 rotating registers
 - R[1], R[2],...R[32]
 - we have four sets of above, allow same references in four different iterations
- CS 211



Rotating Loop Frames for Loop Pipelining

Suppose B_i is only data dependent (through data stored in registers) on A_i ; and C_i only on B_i .

- The "pipelined" kernel block (containing independent computation from C_i , B_{i+1} and A_{i+2}) potentially has better ILP

What happens if C_i is also data dependent on A_i

- The result placed in register by A gets clobbered by the next execution of A (in the next cycle) before C can use it two cycles from now

CS 211

Nice Loop Pipelining Example

```

i=0
while (i<99) {
  :: a[i]=a[i]/10
  Rx = a[i]
  Ry = Rx / 10
  a[i] = Ry
  i++
}

```

→

```

i=0
while (i<99) {
  Rx = a[i]
  Ry = Rx / 10
  a[i] = Ry
  Rx = a[i+1]
  Ry = Rx / 10
  a[i+1] = Ry
  Rx = a[i+2]
  Ry = Rx / 10
  a[i+2] = Ry
  i=i+3
}

```

→

```

i=0
Ry=a[0] / 10
Rx=a[1]
while (i<97) {
  a[i]=Ry
  Ry=Rx / 10
  Rx=a[i+2]
  i++
}
a[97]=Ry
a[98]=Rx / 10

```

CS 211

Loop Pipelining Requiring Renaming

```

i=0
while (i<99) {
  :: a[i]=a[i]/10+a[i]
  Rx = a[i]
  Ry = Rx / 10
  a[i] = Ry+Rx
  i++
}

```

→

```

i=0
while (i<99) {
  Rx = a[i]
  Ry = Rx / 10
  a[i] = Ry+Rx
  Rx = a[i+1]
  Ry = Rx / 10
  a[i+1] = Ry+Rx
  Rx = a[i+2]
  Ry = Rx / 10
  a[i+2] = Ry+Rx
  i=i+3
}

```

→

```

i=0
Ry=a[0] / 10
Rx=a[1]
while (i<97) {
  a[i]=Ry+Rx'
  Ry=Rx / 10
  Rx'=Rx
  Rx=a[i+2]
  i++
}
a[97]=Ry + Rx'
a[98]=Rx / 10 + Rx

```

CS 211

Renaming with Rotating Registers

```

i=0
Ry=a[0] / 10
Rx=a[1]
while (i<97) {
  a[i]=Ry+Rx'
  Ry=Rx / 10
  Rx'=Rx
  Rx=a[i+2]
  i++
}
a[97]=Ry + Rx'
a[98]=Rx / 10 + Rx

```

→

```

i= -2
while (i<99) {
  pred(i>=1):
    a[i]=Ry+RR(x-2)
  pred(i>=-2 && i<98):
    Ry=RR(x-1) / 10
  pred(i<97):
    RR(x)=a[i+2]
  `increase RR offset by 1'
  i++
}

```

CS 211

Register Renaming

- 128 general purpose physical integer registers
- Register names R0 to R31 are static and refer to the first 32 physical GPRs
- Register names R32 to R127 are known as “rotating registers” and are renamed onto the remaining 96 physical registers by an offset
- Remapping wraps around the rotating registers such that when offset is non-zero, physical location of R127 is just below R32

IA-64 Architecture

- 128 general-purpose registers
- 128 floating-point registers
- Arbitrary number of functional units
- Arbitrary latencies on the functional units
- Arbitrary number of memory ports
- Arbitrary implementation of the memory hierarchy

Needs retargetable compiler and recompilation to achieve maximum program performance on different IA-64 implementations

CS 211

IA-64 Instruction Format

- **IA-64 “Bundle”**
 - Total of 128 bits
 - Contains three IA-64 instructions (*aka syllables*)
 - Template bits in each bundle specify dependencies both within a bundle as well as between sequential bundles
 - A collection of independent bundles forms a “group”

A more efficient and flexible way to encode ILP than a fixed VLIW format

<i>inst₁</i>	<i>inst₂</i>	<i>inst₃</i>	<i>temp</i>
-------------------------	-------------------------	-------------------------	-------------
- **IA-64 Instruction**
 - Fixed-length 40 bits long
 - Contains three 7-bit register specifiers
 - Contains a 6-bit field for specifying one of the 64 one-bit predicate registers

CS 211

IA-64 EPIC vs. Classic VLIW

- **Similarities:**
 - Compiler generated wide instructions
 - Static detection of dependencies
 - ILP encoded in the binary (a group)
 - Large number of architected registers
- **Differences:**
 - Instructions in a bundle can have dependencies
 - Hardware interlock between dependent instructions
 - Accommodates varying number of functional units and latencies
 - Allows dynamic scheduling and functional unit binding

Static scheduling are “suggestive” rather than absolute

⇒ **Code compatibility across generations**

but software won’t run at top speed until it is recompiled so “shrink-wrap binary” might need to include multiple builds

CS 211



Cool Features of IA64

- Predicated execution
- Speculative, non-faulting Load instruction
- Software-assisted branch prediction
- Register stack
- Rotating register frame
- Software-assisted memory hierarchy

Mostly adapted from mechanisms that had existed for VLIWs

CS 211



Itanium Specifics

- 6-wide 10-stage pipeline
- Fetch 2 bundles per cycle with the help of BP into a 8-bundle deep fetch queue
- 512-entry 2-level BPT, 64-entry BTAC, 4 TAR, and a RSB
- Issue up to 2 bundles per cycle some mixes of 6 instructions e.g. (MFI,MFI) or (MIB,MIB_i)
- Can issue as little as one syllable per cycle on RAW hazard interlock or structural hazard (scoreboard for RAW detection)
- 8R-6W 128 Entry Int. GPR, 128 82-bit FPR, 64 predicate reg's
- 4 globally-bypassed single-cycle integer ALUs with MMX, 2 FMACs, 2 LSUs, 3 BUs
- Can execute IA-32 software directly

- Intended for high-end server and workstations
- You can buy one now, finally.

CS 211



Introduction to Optimizing Compilers

CS 211



EPIC and Compiler Optimization

- EPIC requires dependency free “scheduled code”
- Burden of extracting parallelism falls on compiler
- success of EPIC architectures depends on efficiency of Compilers!!
- We provide overview of Compiler Optimization techniques (as they apply to EPIC/LP)

CS 211

