

---

# EECS 452 – Lecture 8

## VLIW

---

Instructor: Gokhan Memik

EECS Dept., Northwestern University

# Independence ISA

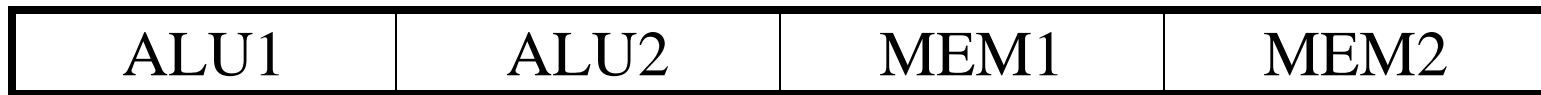


- Conventional ISA
  - Instructions execute in order
- No way of stating
  - Instruction A is **independent** of B
- Vectors and SIMD
  - Only for a set of the same operation
- Idea:
  - Change Execution Model at the ISA model
  - Allow specification of independence
- Goals:
  - Flexible enough
  - Match well technology
- These are some of the goals of VLIW



- **Very Long Instruction Word**

Instruction format



- **#1 defining attribute**
  - The four instructions are *independent*
- Some parallelism can be expressed this way
- Extending the ability to specify parallelism
  - Take into consideration technology
  - Recall, delay slots
  - This leads to →
- **#2 defining attribute: NUAL**
  - **Non-unit assumed latency**

# NUAL vs. UAL



## ■ Unit Assumed Latency (UAL)

- Semantics of the program are that each instruction is completed before the next one is issued
- This is the conventional sequential model

## ■ Non-Unit Assumed Latency (NUAL):

- At least 1 operation has a non-unit assumed latency,  $L$ , which is greater than 1
- The semantics of the program are correctly understood if exactly the next  $L-1$  instructions are understood to have issued before this operation completes

# #2 Defining Attribute: NUAL



## ■ Assumed latencies for all operations

Instruction format

ALU1	ALU2	MEM1	control
ALU1	ALU2	MEM1	control
ALU1	ALU2	MEM1	control
ALU1	ALU2	MEM1	control
ALU1	ALU2	MEM1	control
ALU1	ALU2	MEM1	control
ALU1	ALU2	MEM1	control
ALU1	ALU2	MEM1	control

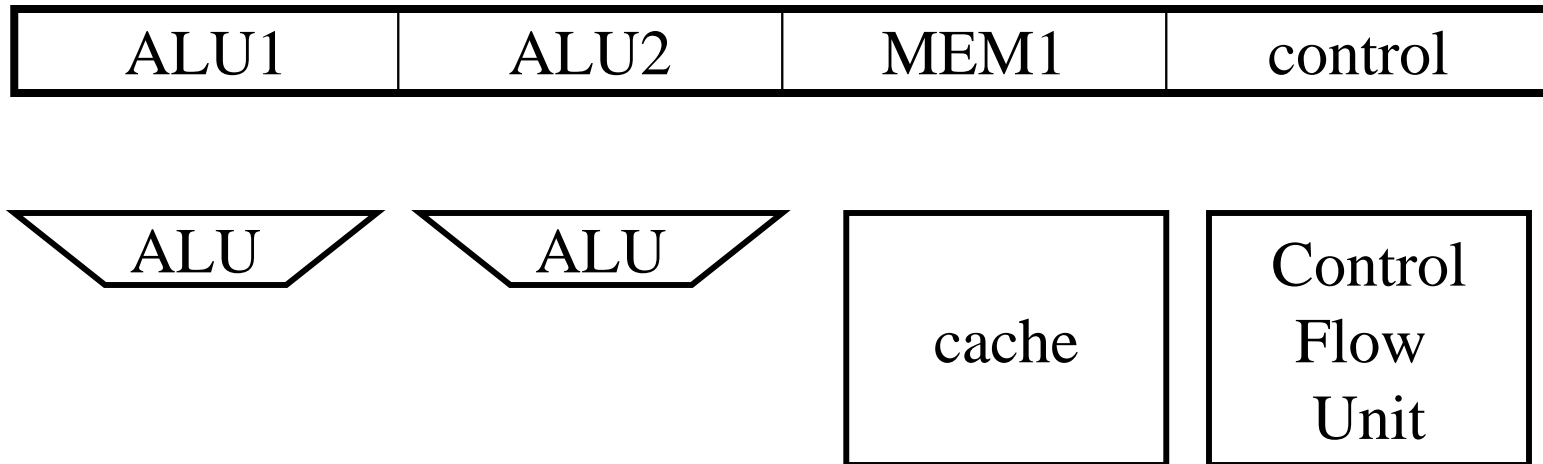
Red arrows point from the top row to the corresponding cells in the bottom row. Blue text 'visible' is placed below the ALU1, ALU2, MEM1, and control cells in the bottom row.

- Glorified delay slots
- Additional opportunities for specifying parallelism

# #3 DF: Resource Assignment



- The VLIW also implies allocation of resources
- This maps well onto the following datapath:

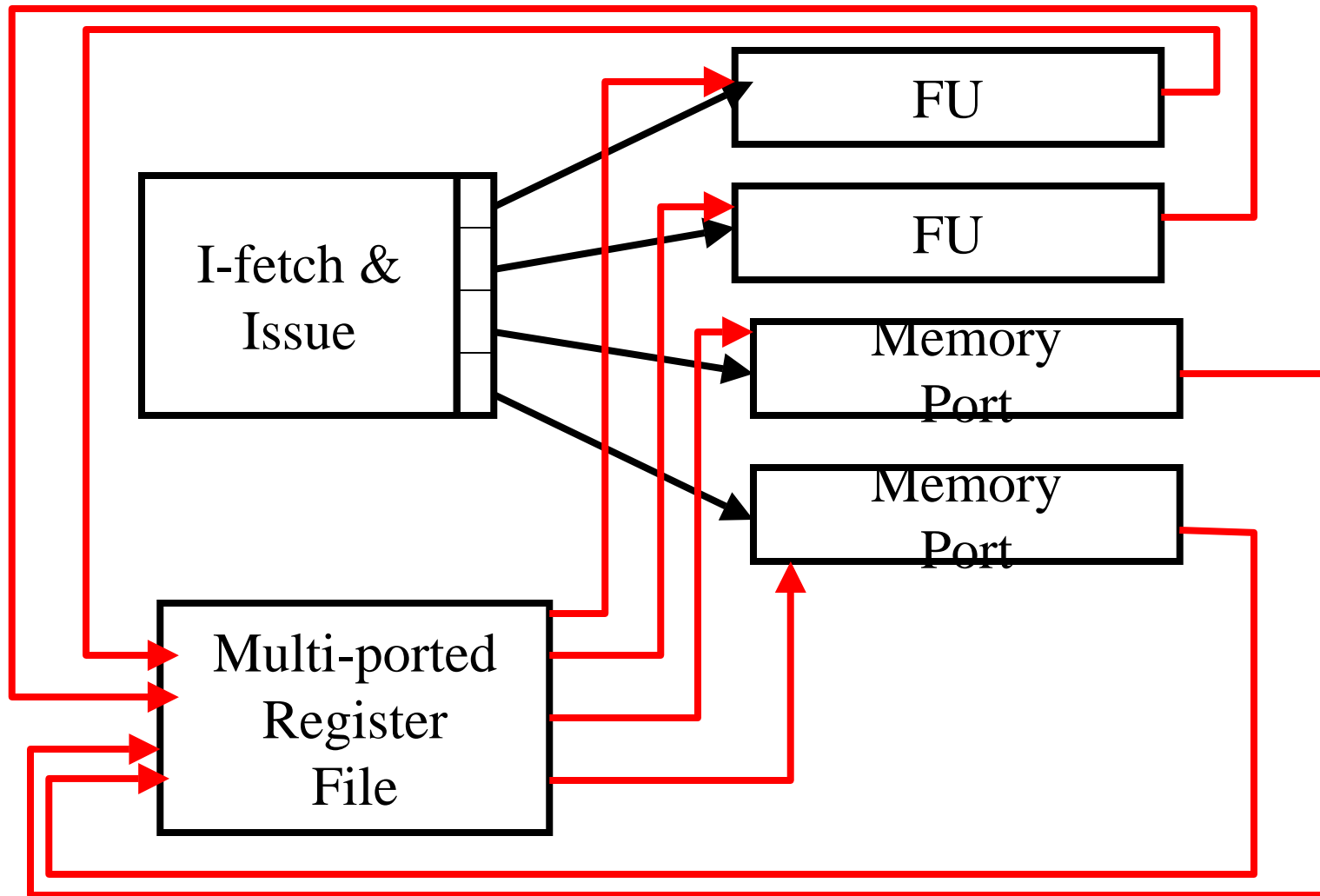


# VLIW: Definition



- Multiple independent Functional Units
- Instruction consists of multiple independent instructions
- Each of them is aligned to a functional unit
- Latencies are fixed
  - Architecturally visible
- Compiler packs instructions into a VLIW also schedules all hardware resources
- Entire VLIW issues as a single unit
- Result: ILP with simple hardware
  - compact, fast hardware control
  - fast clock

# VLIW Example



# VLIW Example



## Instruction format

ALU1	ALU2	MEM1	MEM2
------	------	------	------

## Program order and execution order

ALU1	ALU2	MEM1	MEM2
ALU1	ALU2	MEM1	MEM2
ALU1	ALU2	MEM1	MEM2

- Instructions in a VLIW are **independent**
- Latencies are fixed in the architecture spec.
- Hardware does not check anything
- Software has to schedule so that all works

# Compilers are King



- VLIW philosophy:
  - “dumb” hardware
  - “intelligent” compiler
  
- Key technologies
  - Predicated Execution
  - Trace Scheduling
    - If-Conversion
  - Software Pipelining

# Predicated Execution



- Instructions are predicated
  - if (cond) then perform instruction
  - In practice
    - calculate result
    - if (cond) destination = result
- *Converts control flow dependences to data dependences*
- if ( a == 0)
  - b = 1;
  - else              b = 2;
  
- 1;          pred = (a == 0)
  - pred;   b = 1
  - !pred;  b = 2

# Predicated Execution: Trade-offs



- Is predicated execution always a win?
- Is predication meaningful for VLIW only?

# If-Conversion



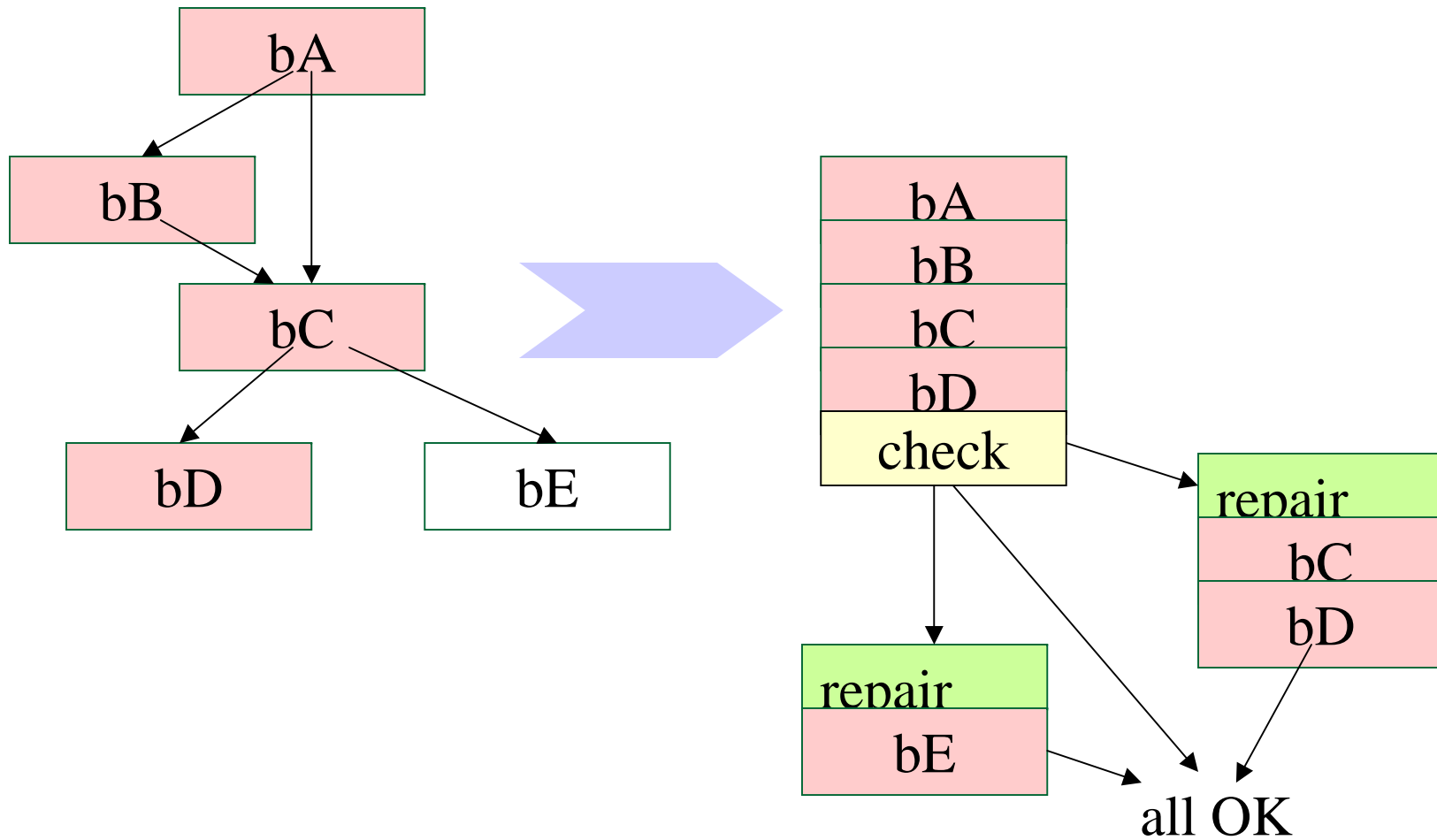
- Late 70's – Early 80's (Ken Kennedy, PLDI 1983)
- Compiler support
  - Early 90's, Mahlke and Hwu, MICRO-92
- Predicate large chunks of code
  - No control flow
- Schedule
  - Free motion of code since no control flow
  - All restrictions are data related
- Reverse if-convert
  - Reintroduce control flow
  - N.J. Warter, S.A. Mahlke, W.W. Hwu, and B.R. Rau. *Reverse if-conversion*. In Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation, pages 290-299, June 1993.

# Trace Scheduling



- Goal:
  - Create a large continuous piece of code
  - Schedule to the max: exploit parallelism
- Fact of life:
  - Basic blocks are small
  - Scheduling across BBs is difficult
- But:
  - while many control flow paths exist
  - There are few “hot” ones
- Trace Scheduling
  - Static control speculation
  - Assume specific path
  - Schedule accordingly
  - Introduce check and repair code where necessary
- First used to compact microcode
  - FISHER, J. *Trace scheduling: A technique for global microcode compaction*. IEEE Transactions on Computers C-30, 7 (July 1981), 478--490.

# Trace Scheduling Example



# Trace Scheduling Example



```
test = a[i] + 20;
```

```
If (test > 0) then
```

```
    sum = sum + 10
```

```
else
```

```
    sum = sum + c[i]
```

```
c[x] = c[y] + 10
```

```
test = a[i] + 20
```

```
sum = sum + 10
```

```
c[x] = c[y] + 10
```

```
if (test <= 0) then goto repair
```

```
...
```

```
repair:
```

```
    sum = sum - 10
```

```
    sum = sum + c[i]
```



Straight code

# Software Pipelining



- Rau, MICRO-81 and Lam, PLDI 88

- A loop

for i = 1 to N

$$a[i] = b[i] + C$$

- Loop Schedule

- 0: LD      f0, 0(r16)
- 1:
- 2:
- 3: ADD     f16, f30, f0
- 4:
- 5:
- 6: ST      f16, 0(r17)
  
- Assume f30 holds C

# Software Pipelining



- Assume latency = 3 cycles for all ops
  - 0:LD f0, 0(r16)
  - 1: LD f1, 8(r16)
  - 2: LD f2, 16(r16)
  - 3:ADD f16, f30, f0
  - 4: ADD f17, f30, f1
  - 5: ADD f18, f30, f2
  - 6:ST f16, 0(r17)
  - 7: ST f17, 8(r17)
  - 8 : ST f18, 16(r17)
- Steady State:
- LD (**i+3**), ADD (**i**),ST (**i – 3**)

# Complete Code



## PROLOG

```
LD f0, 0(r16)
LD f1, 8(r16)
LD f2, 16(r16)
ADD f0,C, f16
ADD f1,C, f17
ADD f2,C, f18
```

## KERNEL

```
ST f16, 0(r17)
ST f17, 8(r17)
ST f18, 16(r17)
ADD f0,C, f16
ADD f1,C, f17
ADD f2,C, f18
LD f0, 0(r16)
LD f1, 8(r16)
LD f2, 16(r16)
```

## EPILOGUE

```
ST f16, 0(r17)
ST f17, 8(r17)
ST f18, 16(r17)
ST f16, 0(r17)
ST f17, 8(r17)
ST f18, 16(r17)
```

# Rotating Register Files Help



ST f19, 0(r17)    ADD f3,C, f16    LD f0, 0(r16)

Special branches

Ctop and wtop

Register references are relative to register base

Branches increment base modulo size

Register file is treated as a circular queue

Finally, predication eliminates prolog and epilogue

(p6) ST f19, 0(r17)

(p3) ADD f3,C, f16

(p0) LD f0, 0(r16)

# VLIW - History



- **Floating Point Systems Array Processor**
  - very successful in 70's
  - all latencies fixed; fast memory
- **Multiflow**
  - Josh Fisher (now at HP)
  - 1980's Mini-Supercomputer
- **Cydrome**
  - Bob Rau
  - 1980's Mini-Supercomputer
- **Tera**
  - Burton Smith
  - 1990's Supercomputer
  - Multithreading
- **Intel IA-64 (Intel & HP)**

# EPIC philosophy



- Compiler creates complete plan of run-time execution
  - At what time and using what resource
  - POE communicated to hardware via the ISA
  - Processor obediently follows POE
  - No dynamic scheduling, out of order execution
    - These second guess the compiler's plan
- Compiler allowed to play the statistics
  - Many types of info only available at run-time
    - branch directions, pointer values
  - Traditionally compilers behave conservatively → handle worst case possibility
  - Allow the compiler to gamble when it believes the odds are in its favor
    - Profiling
- Expose micro-architecture to the compiler
  - memory system, branch execution

# Defining feature I - MultiOp



## ■ Superscalar

- Operations are sequential
- Hardware figures out resource assignment, time of execution

## ■ MultiOp instruction

- Set of independent operations that are to be issued simultaneously
  - no sequential notion within a MultiOp
- 1 instruction issued every cycle
  - Provides notion of time
- Resource assignment indicated by position in MultiOp
- POE communicated to hardware via MultiOps

# Defining feature II - Exposed



latency

## ■ Superscalar

- ❑ Sequence of atomic operations
- ❑ Sequential order defines semantics (UAL)
- ❑ Each conceptually finishes before the next one starts

## ■ EPIC – non-atomic operations

- ❑ Register reads/writes for 1 operation separated in time
- ❑ Semantics determined by relative ordering of reads/writes

## ■ Assumed latency (NUAL if $> 1$ )

- ❑ Contract between the compiler and hardware
- ❑ Instruction issuance provides common notion of time

# EPIC Architecture Overview



- Many specialized registers
  - 32 Static General Purpose Registers
  - 96 Stacked/Rotated GPRs
    - 64 bits
  - 32 Static FP regs
  - 96 Stacked/Rotated FPRs
    - 81 bits
  - 8 Branch Registers
    - 64 bits
  - 16 Static Predicates
  - 48 Rotating Predicates



- 128-bit Instruction Bundles
- Contains 3 instructions
- 6-bit template field
  - FUs instructions go to
  - Termination of independence bundle
  - WAR allowed within same bundle
  - Independent instruction may spread over multiple bundles



# Other architectural features of EPIC

- Add features into the architecture to support EPIC philosophy
  - Create more efficient POEs
  - Expose the microarchitecture
  - Play the statistics
- **Register structure**
- **Branch architecture**
- **Data/Control speculation**
- **Memory hierarchy**
- **Predicated execution**
  - largest impact on the compiler

# Register Structure



## ■ Superscalar

- Small number of architectural registers
- Rename using large pool of physical registers at run-time

## ■ EPIC

- Compiler responsible for all resource allocation including registers
- Rename at compile time
  - large pool of regs needed

# Rotating Register File

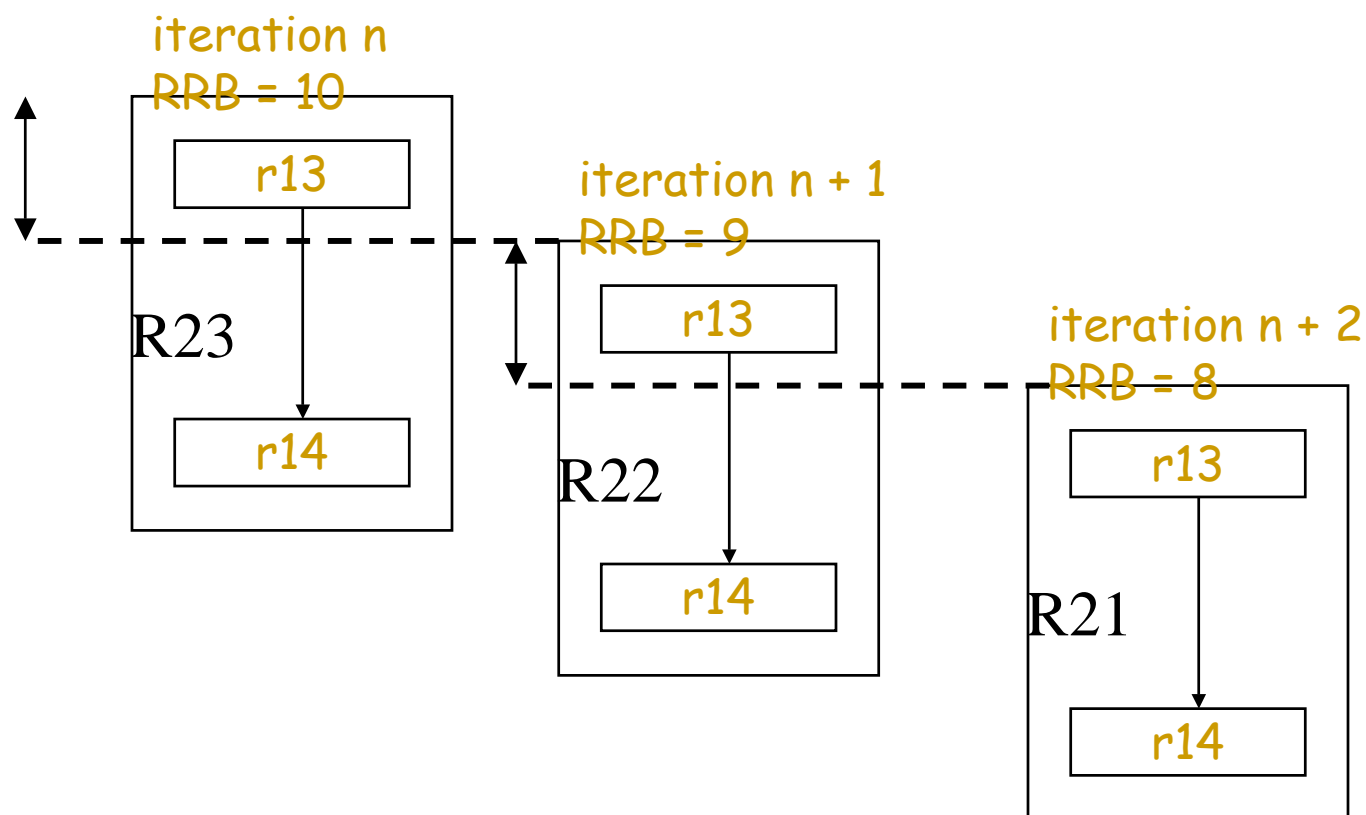


- **Overlap loop iterations**
  - How do you prevent register overwrite in later iterations?
  - Compiler-controlled dynamic register renaming
  
- **Rotating registers**
  - Each iteration writes to r13
  - But this gets mapped to a different physical register
  - Block of consecutive regs allocated for each reg in loop corresponding to number of iterations it is needed



# Rotating Register File Example

- actual reg = (reg + RRB) % NumRegs
- At end of each iteration, RRB--



# Branch Architecture



## ■ Branch actions

- ❑ Branch condition computed
- ❑ Target address formed
- ❑ Instructions fetched from taken, fall-through or both paths
- ❑ Branch itself executes
- ❑ After the branch, target of the branch is decoded/executed

## ■ Superscalar processors use hardware to hide the latency of all the actions

- ❑ Icache prefetching
- ❑ Branch prediction – Guess outcome of branch
- ❑ Dynamic scheduling – overlap other instructions with branch
- ❑ Reorder buffer – Squash when wrong

# EPIC Branches



- Make each action visible with an architectural latency
  - No stalls
  - No prediction necessary (though sometimes still used)
- Branch separated into 3 distinct operations
  - 1. Prepare to branch
    - compute target address
    - Prefetch instructions from likely target
    - Executed well in advance of branch
  - 2. Compute branch condition – comparison operation
  - 3. Branch itself
- Branches with latency  $> 1$ , have delay slots
  - Must be filled with operations that execute regardless of the direction of the branch

# Predication



```
If a[i].ptr != 0
    b[i] = a[i].left;
else
    b[i] = a[i].right;
i++
```

## Conventional

```
load a[i].ptr
p2 = cmp a[i].ptr != 0
Jump if p2 nodecr
load r8 = a[i].left
store b[i] = r8
jump next
nodecr:
load r9 = a[i].right
store b[i] = r9

next:
i++
```

## IA-64

```
load a[i].ptr
p1, p2 = cmp a[i].ptr != 0
<p1> load a[i].l      <p2> load.a[i].r
<p1> store b[i], r8  <p2> store b[i], r9
i++
```

# Speculation



- Allow the compiler to play the statistics
  - Reordering operations to find enough parallelism
  - Branch outcome
    - Control speculation
  - Lack of memory dependence in pointer code
    - Data speculation
  - Profile or clever analysis provides “the statistics”
- General plan of action
  - Compiler reorders aggressively
  - Hardware support to catch times when its wrong
  - Execution repaired, continue
    - Repair is expensive
    - So have to be right most of the time to or performance will suffer

# “Advanced” Loads



```
t1=t1+1
if (t1 > t2)
    j = a[t1 + t2]
```

```
add t1 + 1
comp t1 > t2
Jump donothing
load a[t1 - t2]
donothing:
```

```
add t1 + 1
ld.s r8=a[t1 - t2]
comp t1>t2
jump[
check.s r8
```

**ld.s**: load and record  
Exception  
**Check.s** check for  
Exception  
Allows load to be  
Performed early

**Not IA-64 specific**

# Speculative Loads



- Memory Conflict Buffer (Illinois)
- Goal: Move load before a store when unsure that a dependence exists
- Speculative load:
  - Load from memory
  - Keep a record of the address in a table
- Stores check the table
  - Signal error in the table if conflict
- Check load:
  - Check table for signaled error
  - Branch to repair code if error
- How are the CHECK and SPEC load linked?
  - Via the target register specifier
- Similar effect to dynamic speculation/synchronization

# Exposed Memory Hierarchy



- Conventional Memory Hierarchies have storage presence speculation mechanism built-in
- Not always effective
  - Streaming data
  - Latency tolerant computations
- EPIC:
  - Explicit control on where data goes to:

Source cache specifier - where its coming from → latency

L\_B\_C3\_C2                      S\_H\_C1

Target cache specifier - where to place the data

- Conventional: C1/C1

# VLIW Discussion



- Can one build a dynamically scheduled processor with a VLIW instruction set?
- VLIW really simplifies hardware?
- Is there enough parallelism visible to the compiler?
  - What are the trade-offs?
- Many DSPs are VLIW
  - Why?