CSE 431
Computer Architecture
Fall 2017

VLIW Datapaths

Mahmut Taylan Kandemir (www.cse.psu.edu/~kandemir)

[Adapted from *Computer Organization and Design, 5th Edition*,

Patterson & Hennessy, © 2014, MK

With additional thanks/credits to Amir Roth, Milo Martin]

Roadmap

- Now: static parallelism
- Next: dynamic parallelism
- After that: memory hierarchy

Review: Pipeline Hazards

- Structural hazards
  - Design pipeline to eliminate structural hazards
- Data hazards – read after write (RAW)
  - Use data forwarding inside the pipeline
  - For those cases that forwarding won't solve (e.g., load-use) include hazard hardware to insert stalls in the instruction stream
- Control hazards – `beq`, `bne,j,jr,jal,jalr`
  - Move decision point as early in the pipeline as possible (e.g., Decode) – reduces number of stalls at the cost of additional hardware
  - Predict – with dynamic branch prediction, can reduce the impact of control hazard flushes even further if the branch prediction (contained in a BHT in the Fetch stage) is correct (shoot for 97%+ accuracy) and if the branched-to instruction is cached (in a BTB also in the Fetch stage)

Extracting Yet *More* Performance

- Increase the depth of the pipeline to increase the clock rate (CPI still 1, IC unchanged) – superpipelining
  - The more stages in the pipeline, the more forwarding/hazard hardware needed and the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time)
- Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – multiple-issue
  - The instruction execution rate, CPI, will be less than 1, so instead we use IPC: instructions per clock cycle
    - E.g., a 6 GHz, four-way multiple-issue core can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4
  - If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?

Types of Code Parallelism

- Instruction-level parallelism (ILP) of a program – a measure of the average number of instructions in a program that a core *might* be able to execute at the same time
  - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions
  - If you had a large enough datapath, ILP would determine the limit on how fast a program could run.

```
DO I = 1 TO 100
   A[I] = A[I] + 1
CONTINUE
```

- Data-level parallelism (DLP)
  - The example shown has lots of data parallelism but almost no instruction parallelism if compiled in the obvious way.
  - Loop unrolling is a common compiler optimization. If completely unrolled, and if we had 100 arithmetic/addressing units and 100 memory ports, we could achieve a speedup of 100 over a scalar core.

Datapath (Core) Parallelism

- Machine-level parallelism of a datapath – a measure of the ability of the datapath to take advantage of the ILP of the program
  - Determined by the number of instructions that can be fetched and executed at the same time
- To achieve high performance, need both instruction-level parallelism and machine-level parallelism
- Some additional examples
  - SIMD instructions, short-vector packed data
  - Multithreading (MT)
  - Simultaneous Multithreading (SMT) (aka as Hyperthreading)
  - Multicore, homogeneous and heterogeneous (GPGPUs)
  - Multiprocessor
  - others?

Multiple Instruction Issue

- Fetch and issues multiple instructions at a time
- Instructions issued at the same cycle should not have dependences between them

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Multiple Instruction Issue Possibilities

- Fetch and issue **more than one** instruction in a cycle

1. **Statically-scheduled (in-order)**
   - **Very Long Instruction Word (VLIW)** e.g., TransMeta (4-wide)
     - Compiler figures out what can be done in parallel, so the hardware can be dumb and low power
     - Compiler must group parallel instr's, requires new binaries
   - **SuperScalar** e.g., Pentium (2-wide), ARM Cortex-A8 (2-wide)
     - Hardware figures out what can be done in parallel
     - Executes unmodified sequential programs
   - **Explicitly Parallel Instruction Computing (EPIC)** e.g., Intel Itanium (6-wide)
     - A compromise: compiler does some, hardware does the rest
2. **Dynamically-scheduled (out-of-order) SuperScalar**
   - Hardware dynamically determines what can be done in parallel (can extract much more ILP with OOO processing)
   - E.g., Intel Core i7 (4-wide, 8-way SMT, 4 cores/chip), IBM Power8 (8-wide, 8-way SMT, 12 cores/chip)

Taxonomy of Multiple-Issue Machines

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - How many instructions to issue in one clock cycle
  - Storage (data) dependencies → data hazards
    - Limitation more severe in an in-order SuperScalar/VLIW core
  - Procedural dependencies → control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
    - Use loop unrolling (in the compiler) to increase ILP and reduce the occurrence of branches
  - Resource conflicts → structural hazards
    - A multiple-issue datapath has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and RF write ports
    - Resource conflicts can often be eliminated by duplicating the resource or by pipelining the resource

Review: A Quick Overview of Dependence Analysis

- To what extent can the compiler (or the datapath) reorder instructions? Are there execution-order constraints?

Original	possible?	possible?

`instr 1`
`instr 2`
consecutive
	`instr 2`
	`instr 1`
	consecutive
		`instr 1` and `instr 2`
		simultaneous

- Instruction dependencies imply that reordering instructions is not possible
  - true dependence (or, data dep., flow dep.) (cannot reorder)

```
a = .
. = a
```
RAW, read after write

  - anti-dependence (renaming allows reordering)

```
. = a
a = .
```
WAR, write after read

  - output dependence (renaming allows reordering)

```
a = .
a = .
```
WAW, write after write

Eliminating Name Dependences

- What is the fundamental difference between data dependence and name dependence?

- Name dependences can be eliminated via Renaming
  - Either in hardware or in software

- Can renaming work for data dependences?

Dependences and Hazards

- A hazard exists whenever
  - there is a name or data dependence between instructions, and
  - they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence
- Because of the dependence, we must preserve what is called program order (semantics of the sequential code)
- The goal of hardware and software is to exploit parallelism by preserving program order
- Therefore, a data dependence conveys three things:
  - the possibility of a hazard
  - the order in which results must be calculated
  - an upper bound on how much parallelism could possibly be exploited
- A dependence can be overcome in two different ways:
  - Maintaining the dependences but avoiding a hazard
  - Eliminating the dependence by transforming the code

Implications of Instruction Dependencies

- Instruction dependencies imply that reordering instructions is not possible
  - control dependence (reordering might be possible, but how and when would you know?)

```
          beq $s0, $s1, Label
          a = .
Label:    . = a
```

  - input "dependence" (reordering is possible)

```
. = a
. = a
```

RAR, read after read (as long as `.` is to a different storage location (in DM or the RF))

  - loop dependence (renaming allows reordering, but how many?)

```
a = .      in one iteration of the loop
a = .      in a later iteration of the loop, the same instruction
```

- Instruction dependencies produce a graph of the program (called dependence graph), which can be analyzed for ILP
  - This must be a conservative analysis – any dependence which cannot be proved non-existent is assumed to exist

Speculation in VLIW Cores

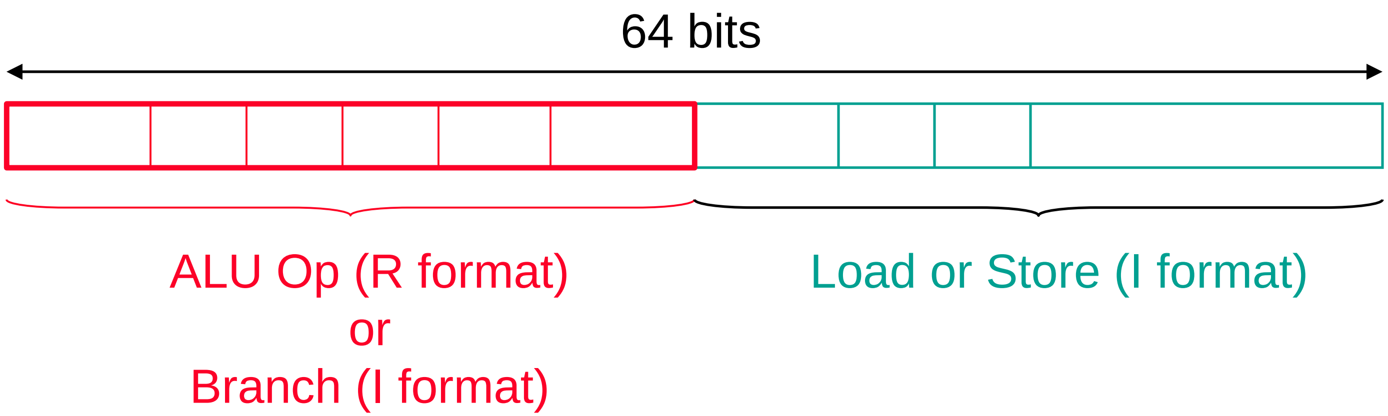
- Speculation is used to allow execution of future instr's that (may) depend on the speculated instruction
  - Speculate on the outcome of a conditional branch (branch prediction)
  - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (load speculation)
- What if the speculation was wrong?
  - In a VLIW core the compiler inserts additional instructions that check the accuracy of the speculation and provides a fix-up routine to use when the speculation was incorrect
- Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur (i.e., not allowed to change the machine state until you are sure)

VLIW Multiple Issue Datapaths

- VLIW multiple-issue datapath has the compiler statically decide which instructions to issue and execute simultaneously
  - Issue packet – the set of instructions that are bundled together and issued in one clock cycle – think of it as one large instruction with multiple operations
    - Compiler guarantees that the instr's within a packet are independent (this usually means some of the instr's in the packet are `nops`)
  - The mix of instructions in the packet (bundle) is usually restricted – a single "instruction" with several predefined, "slotted" fields
  - The compiler does static branch prediction and code scheduling (with renaming) to reduce control hazards and eliminate WAW & WAR data hazards
- VLIWs have
  - Multiple functional units
  - Multi-ported register files
  - Wide program bus

An Example: A VLIW MIPS

- Consider a 2-wide issue MIPS with a 2-instr packet that is fetched, decoded and issued for execution as a pair



- Data hazards ?
  - Load-use (RAW) hazards have to be split by the compiler into two packets (since there is no way to forward if they are happening simultaneously)

```
lw    $2, 400($3)
add   $1, $2, $4
```

```
add   $1, $2, $4
lw    $2, 400($3)
```

Working Around Structural Hazards

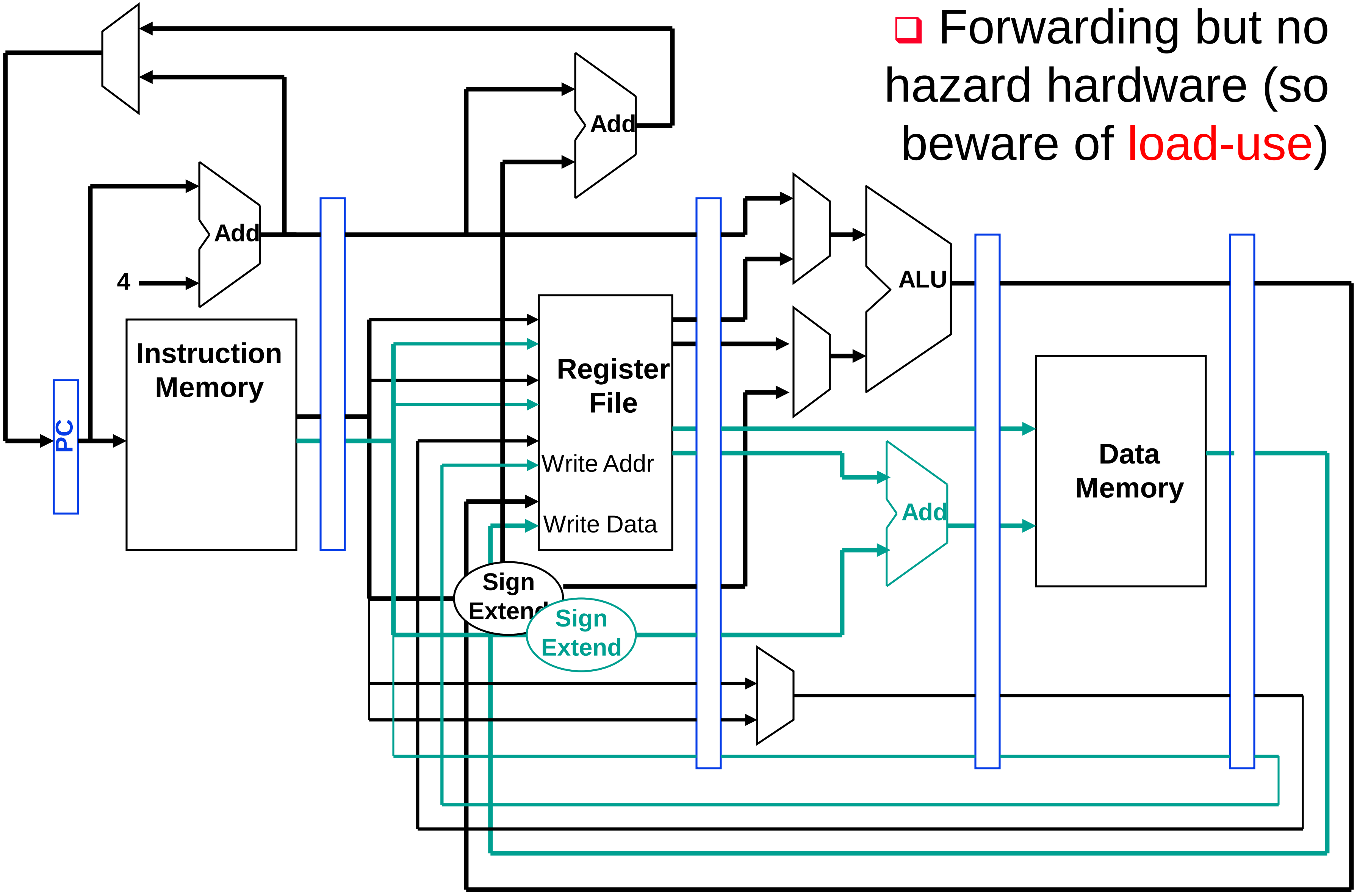
- FETCH – 2-wide issue packets so fetch a 64-bit "instruction" packet (cache blocks of at least 8B)

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

- ID (and WB) – Need a 4 read port and 2 write port RF and two decoders
- EX – Need a separate memory address adder
- MEM – Only one of the pair touches data memory (so no structural hazard there)

A MIPS VLIW (2-issue) Pipelined Datapath

- Forwarding but no hazard hardware (so beware of load-use)



Code Scheduling Example

- Consider the following loop code

```
lp:   lw    $t0,0($s1)     # $t0=array element
      addu  $t0,$t0,$s2    # add scalar in $s2
      sw    $t0,0($s1)     # store result
      addi  $s1,$s1,-4     # decrement pointer
      bne   $s1,$0,lp      # branch if $s1 != 0
```

- Compiler "schedules" the instr's to avoid pipeline stalls
  - Instructions in one bundle *must* be independent
  - Must separate load-use instructions from their loads by one cycle (we are assuming a use latency of **one** cycle)
  - Notice that the first two instructions have a load-use dependency (in red), the next two and last two have true (RAW) data dependencies (in blue and green)
  - Assume branches are perfectly predicted by the hardware

The Scheduled Code (Not Unrolled)

	ALU or branch	Data transfer	CC
`lp:`		`lw $t0,0($s1)`	1
	`addi $s1,$s1,-4`		2
	`addu $t0,$t0,$s2`		3
	`bne $s1,$0,lp`	`sw $t0,4($s1)`	4
			5

- 4 clock cycles to execute 5 instructions for a
  - CPI of 0.8 (versus the best case of 0.5)
  - IPC of 1.25 (versus the best case of 2.0)
  - Three `nops` (that don’t count towards performance !!)

Need for Loop Unrolling

- Compiler needs more instructions to fill empty slots
  - One loop iteration does not provide enough operations
  - We need more (useful) operations relative to the number of overhead instructions
  - This is where “loop unrolling” comes to the picture
- How do we achieve this?
  - Borrow operations from other loop iterations
  - This is what loop unrolling enables

Loop Unrolling

- Compiler loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP
- Apply loop unrolling (4 times for our example) and then schedule the resulting code
  - Reduces loop-control overhead
  - Schedule instr's so as to avoid load-use and other RAW hazards
  - Schedule instr's so as to obey loop-carried dependencies (RAW), e.g., store in one loop followed by a load of the same register in the next loop
- During unrolling the compiler applies register renaming to eliminate all data dependencies that are not true data dependencies, i.e., WAW and WAR data hazards

Unrolled Code Example

```
lp:  lw    $t0,0($s1)     # $t0=array element
     lw    $t1,-4($s1)    # $t1=array element
     lw    $t2,-8($s1)    # $t2=array element
     lw    $t3,-12($s1)   # $t3=array element
     addu  $t0,$t0,$s2    # add scalar in $s2
     addu  $t1,$t1,$s2    # add scalar in $s2
     addu  $t2,$t2,$s2    # add scalar in $s2
     addu  $t3,$t3,$s2    # add scalar in $s2
     sw    $t0,0($s1)     # store result
     sw    $t1,-4($s1)    # store result
     sw    $t2,-8($s1)    # store result
     sw    $t3,-12($s1)   # store result
     addi  $s1,$s1,-16    # decrement pointer
     bne   $s1,$0,lp      # branch if $s1 != 0
```

The Scheduled Code (Unrolled) … almost

	ALU or branch	Data transfer	CC
`lp:`		`lw $t0,0($s1)`	1
		`lw $t1,-4($s1)`	2
		`lw $t2,-8($s1)`	3
		`lw $t3,-12($s1)`	4
		`sw $t0,0($s1)`	5
		`sw $t1,-4($s1)`	6
		`sw $t2,-8($s1)`	7
		`sw $t3,-12($s1)`	8

- First schedule the data transfers (which can't be done in less than 8 cycles)
  - Notice the abundant use of registers

The Scheduled Code (Unrolled) … almost

	ALU or branch	Data transfer	CC
`lp:`		`lw $t0,0($s1)`	1
		`lw $t1,-4($s1)`	2
	`addu $t0,$t0,$s2`	`lw $t2,-8($s1)`	3
	`addu $t1,$t1,$s2`	`lw $t3,-12($s1)`	4
	`addu $t2,$t2,$s2`	`sw $t0,0($s1)`	5
	`addu $t3,$t3,$s2`	`sw $t1,-4($s1)`	6
		`sw $t2,-8($s1)`	7
	`bne $s1,$0,lp`	`sw $t3,-12($s1)`	8

- Next schedule the data use instr’s in 4 cycles, being sure to leave (at least) one cycle between the load and its data use

- And schedule the branch instr in the last slot

The Scheduled Code (Unrolled) … almost

	ALU or branch	Data transfer	CC
`lp:`		`lw $t0,0($s1)`	1
		`lw $t1,-4($s1)`	2
	`addu $t0,$t0,$s2`	`lw $t2,-8($s1)`	3
	`addu $t1,$t1,$s2`	`lw $t3,-12($s1)`	4
	`addu $t2,$t2,$s2`	`sw $t0,0($s1)`	5
	`addu $t3,$t3,$s2`	`sw $t1,-4($s1)`	6
		`sw $t2,-8($s1)`	7
	`bne $s1,$0,lp`	`sw $t3,-12($s1)`	8

- One final instruction to schedule, where do we put

```
addi   $s1,$s1,-16
```

The Scheduled Code (Unrolled)

	ALU or branch	Data transfer	CC
`lp:`	`addi $s1,$s1,-16`	`lw $t0,0($s1)`	1
		`lw $t1,12($s1)`	2
	`addu $t0,$t0,$s2`	`lw $t2,8($s1)`	3
	`addu $t1,$t1,$s2`	`lw $t3,4($s1)`	4
	`addu $t2,$t2,$s2`	`sw $t0,16($s1)`	5
	`addu $t3,$t3,$s2`	`sw $t1,12($s1)`	6
		`sw $t2,8($s1)`	7
	`bne $s1,$0,lp`	`sw $t3,4($s1)`	8

- Notice the adjustment in the memory address offsets
- Eight clock cycles to execute 14 instructions for a
  - CPI of 0.57 (versus the best case of 0.5)
  - IPC of 1.8 (versus the best case of 2.0), but at the cost of code size and more register use

Steps for Loop Unrolling and Scheduling

- Determine that unrolling the loop would be useful (How?)
- Use different registers to avoid unnecessary constraints (What if not done?)
- Eliminate extra test and branch instructions
- Schedule the code

Loop unrolling and scheduling are complementary!

Unknown number of loop iterations?

- Number of iterations = *n*
- Goal: make *k* copies of the loop body
- Generate pair of loops:
  - First executes *n* mod *k* times
  - Second executes *n* / *k* times
  - Add checks before executing each loop

Compiler Support for VLIW Cores

- The compiler packs groups of independent instructions into the bundle
  - Done by code re-ordering
- The compiler uses loop unrolling to expose more ILP
  - Loop unrolling also reduces the number of conditional branches
- The compiler uses register renaming to solve name dependencies (WAR [anti] and WAW [output]) and ensures no load-use hazards occur by scheduling load-use instr's appropriately
- VLIWs primarily depend on the compiler for branch prediction
- The compiler predicts memory bank references to help minimize memory bank conflicts

VLIW Advantages and Disadvantages

- Advantages
  - Simpler hardware (potentially less power hungry)
    - Compiler does most of the work
  - Potentially more scalable
    - Allow more instr's per VLIW bundle and add more FUs
- Disadvantages
  - Programmer/compiler complexity and longer compilation times
    - Deep pipelines can be confusing as to what can be handled with forwarding and what needs to be stalled
  - Lock step operation, i.e., one slow can stall the entire packet
  - Object (binary) code incompatibility
  - Needs lots of program memory bandwidth
  - Code bloat
    - `nops` are a waste of program memory space
    - Loop unrolling to expose more ILP uses more program memory space as well

VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel

- RISC (John Cocke, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple

- VLIW (Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design

Track Record of VLIWs

- Started with "horizontal microcode"
- Academic projects
  - Yale ELI-512 [Fisher, '85]
  - Illinois IMPACT [Hwu, '91]
- Commercial attempts
  - Multiflow [Colwell+Fisher, '85] → failed
  - Cydrome [Rau, '85] → failed
  - Motorola, TI, … embedded (DSP) cores → successful
    - TI TMS320C62xx
    - Lucent/Motorola StarCoreSC140
  - Intel Itanium [Fisher+Rau, '97] → ??
    - http://en.wikipedia.org/wiki/Itanium
  - Transmeta Crusoe [Ditzel, '99] → mostly failed
    - http://en.wikipedia.org/wiki/Transmeta