CS3350B
Computer Architecture
Winter 2015

Lecture 6.3: Instructional Level Parallelism: Advanced Techniques

Marc Moreno Maza

www.csd.uwo.ca/Courses/CS3350b

[Adapted from lectures on *Computer Organization and Design*, Patterson & Hennessy, 5th edition, 2011]

Greater Instruction-Level Parallelism

- Deeper pipeline (more #stages: 5 => 10 => 15 stages)
  - Less work per stage ⇒ shorter clock cycle
- **Multiple issue** “superscalar”
  - Replicate pipeline stages ⇒ multiple pipelines
    - e.g., have two ALUs or a register file with 4 read ports and 2 write ports
    - have logic to issue several instructions concurrently
  - Execute more than one instruction at a clock cycle, producing an effective CPI < 1, so use **Instructions Per Cycle (IPC)**
  - e.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - If a datapath has a 5-stage pipeline, how many instructions are active in the pipeline at any given time?
  - But dependencies reduce this in practice

Pipeline Depth and Issue Width

- Intel Processors over Time

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Cores	Power
i486	1989	25 MHz	5	1	1	5W
Pentium	1993	66 MHz	5	2	1	10W
Pentium Pro	1997	200 MHz	10	3	1	29W
P4 Willamette	2001	2000 MHz	22	3	1	75W
P4 Prescott	2004	3600 MHz	31	3	1	103W
Core 2 Conroe	2006	2930 MHz	14	4	2	75W
Core 2 Yorkfield	2008	2930 MHz	16	4	4	95W
Core i7 Gulftown	2010	3460 MHz	16	4	6	130W

Multiple-Issue Processor Styles

- **Static** multiple-issue processors, aka **VLIW** (very-long instruction word)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - e.g. Intel Itanium and Itanium 2
    - 128-bit “bundles” containing three instructions
    - Five functional units (IntALU, Mmedia, Dmem, FPALU, Branch)
    - Extensive support for **speculation** and **predication**
- **Dynamic** multiple-issue processors (aka SuperScalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8) are being made dynamically (at run time by the hardware)
    - e.g., IBM power series, Pentium 4, MIPS R10K, AMD Barcelona

Multiple-Issue Datapath Responsibilities

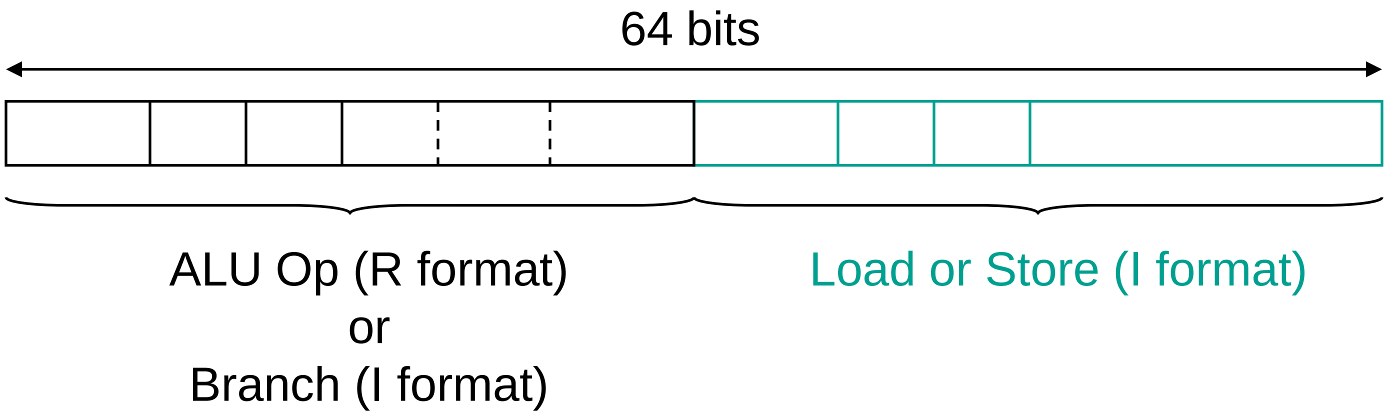
- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - How many instructions to issue in one clock cycle – **issue slots**
  - Storage (data) dependencies – aka **data hazards**
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka **control hazards**
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka **structural hazards**
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

Static Multiple Issue Machines (VLIW)

- Static multiple-issue processors (aka VLIW) use the **compiler** (at compile-time) to 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
  - The mix of instructions in the packet (bundle) is usually restricted – a single "instruction" with several predefined fields
  - The compiler does **static branch prediction** and **code scheduling** to reduce (control) or eliminate (data) hazards
- VLIW's have
  - Multiple functional units
  - Multi-ported register files
  - Wide program bus

An Example: A VLIW MIPS

- Consider a 2-issue MIPS with a 2 instr bundle



- Instructions are always fetched, decoded, and issued in **pairs**
  - If one instr of the pair can not be used, it is replaced with a **nop**

- Need 4 read ports and 2 write ports and a separate memory address adder

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
```

```
/* increment  each element (unsigned integer) in array A by n    */
for (i=m; i>=0; --i)         /* m is the initial value of $s1    */
    A[i] += n;               /* n is the value in register $s2 */
```

- Must “schedule” the instructions to **avoid pipeline stalls**
  - Instructions in one bundle must be **independent**
  - Must separate **load/use** instructions from their loads by one cycle
  - Notice that the first two instructions have a **load/use dependency**, the next two and last two have **data dependencies**
  - Assume branches are perfectly predicted by the hardware

The Scheduled Code (Not Unrolled)

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

```
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
```

- Four 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?)
  - noops don’t count towards performance !!

Loop Unrolling

- 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
  - Eliminate unnecessary **loop overhead** instructions
  - Schedule so as to avoid **load use hazards**

- During unrolling the compiler applies register renaming to eliminate all data dependencies that are not true data dependencies

Loop Unrolling in C

```
for (i=m; i>=0; --i)
    A[i] += n;
```

Assume size of A is 8, i.e. m=7.

Execute not-unrolled code:

Iteration #	i	Instruction
1	7	A[7] += n
2	6	A[6] += n
3	5	A[5] += n
4	4	A[4] += n
5	3	A[3] += n
6	2	A[2] += n
7	1	A[1] += n
8	0	A[0] += n

```
/* unrolled 4 times */
for (i=m; i>=0; i-=4){
   A[i]   += n;
   A[i-1] += n;
   A[i-2] += n;
   A[i-3] += n; }
```

Execute unrolled code:

```
Iteration #1, i=7:
   { A[7] += n;
     A[6] += n;
     A[5] += n;
     A[4] += n; }

Iteration #2, i=3:
   { A[3] += n;
     A[2] += n;
     A[1] += n;
     A[0] += n; }
```

Apply Loop Unrolling for 4 times

```
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
```

```
/* code in c */
for(i=m;i>=0;i-=4)
{
    A[i]   += n;
    A[i-1] += n;
    A[i-2] += n;
    A[i-3] += n;
}
```

```
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
```

- Why not reuse $t0 but use $t1, $t2, $t3?
- Why -4,-8,-12 and $s1=$s1-16?
- How many times can a loop be unrolled?

The Scheduled Code (Unrolled)

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

```
/* code in c */
for(i=m;i>=0;i-=4)
{
   A[i]   += n;
   A[i-1] += n;
   A[i-2] += n;
   A[i-3] += n;
}
```

- 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)

Summary of Compiler Support for VLIW Processors

- The compiler packs groups of independent instructions into the bundle
  - Done by **code re-ordering** (trace scheduling)
- The compiler uses **loop unrolling** to expose more ILP
- The compiler uses **register renaming** to solve name dependencies and ensures **no load use hazards** occur
- While superscalars use dynamic prediction, VLIW's primarily depend on the compiler for **branch prediction**
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-then-else branch structures by replacing them with predicated instructions
- The compiler predicts **memory bank references** to help minimize memory bank conflicts

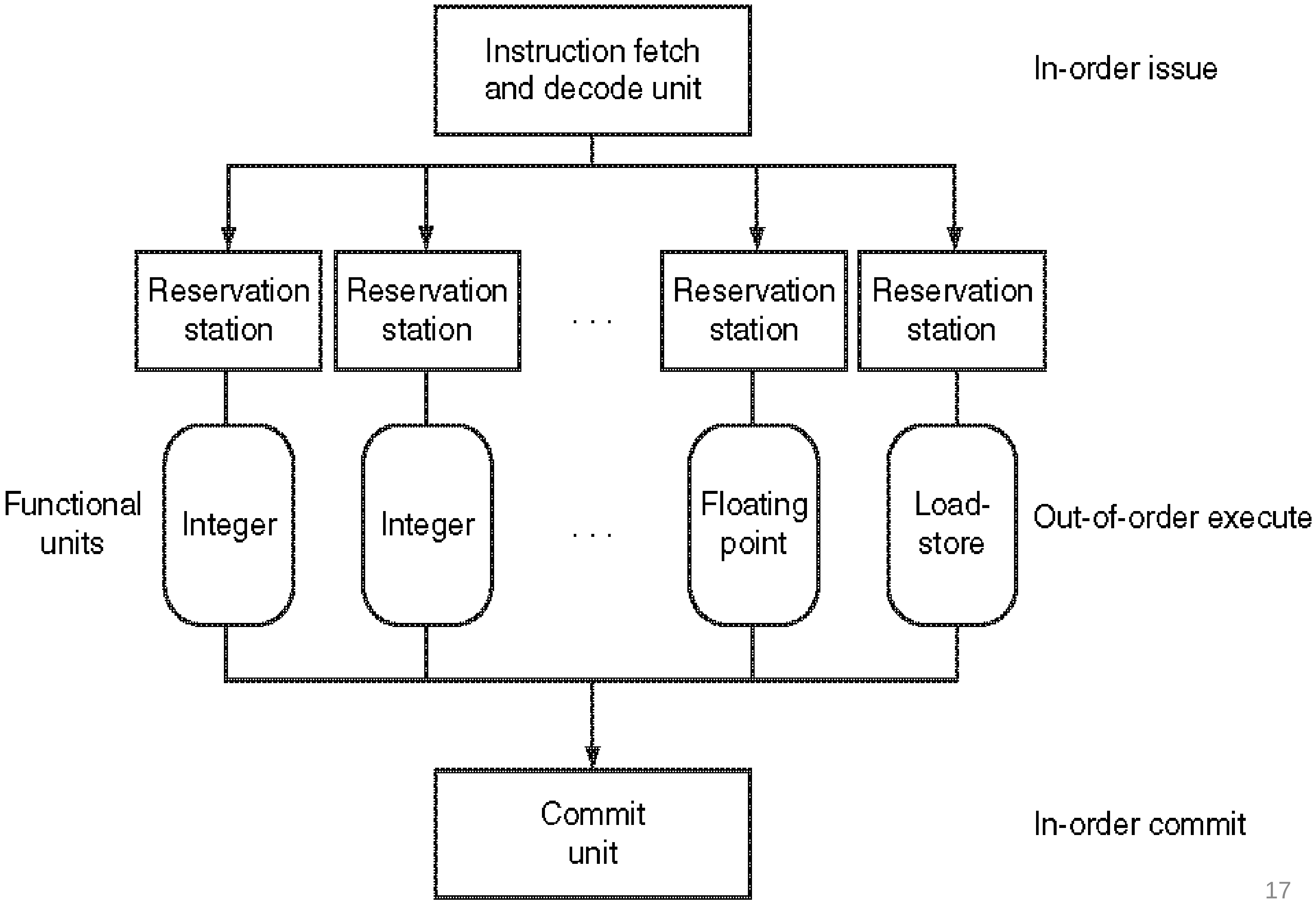
VLIW Advantages & Disadvantages

- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
    - Allow more instr's per VLIW bundle and add more FUs
- Disadvantages
  - Programmer/compiler complexity and longer compilation times
    - Deep pipelines and long latencies can be confusing (making peak performance elusive)
  - Lock step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
  - Object (binary) code incompatibility
  - Needs lots of program memory bandwidth
  - Code bloat
    - Noops are a waste of program memory space
    - Loop unrolling to expose more ILP uses more program memory space

Dynamic Multiple Issue Machines (SS)

- Dynamic multiple-issue processors (aka SuperScalar) use hardware at run-time to dynamically decide which instructions to issue and execute simultaneously
- Instruction-fetch and issue – fetch instructions, decode them, and **issue** them to a FU to await execution
- Instruction-execution – as soon as the source operands and the FU are ready, the result can be calculated
- Instruction-commit – when it is safe to, write back results to the RegFile or D$ (i.e., change the machine state)

Dynamic Multiple Issue Machines (SS)



Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions **out of order** to avoid stalls
  - But commit result to registers in order
- Example

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
subu  $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start `subu` while `addu` is waiting for `lw`

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
  - Disadvantages of complier scheduling code
- Not all stalls are predicable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome (**Branch Prediction**)
    - Roll back if path taken is different
  - Speculate on **load**
    - Roll back if location is updated

Out Of Order Intel

- All use OOO since 2001

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	**No**	1	5W
Pentium	1993	66MHz	5	2	**No**	1	10W
Pentium Pro	1997	200MHz	10	3	**Yes**	1	29W
P4 Willamette	2001	2000MHz	22	3	**Yes**	1	75W
P4 Prescott	2004	3600MHz	31	3	**Yes**	1	103W
Core	2006	2930MHz	14	4	**Yes**	2	75W
Core 2 Yorkfield	2008	2930MHz	16	4	**Yes**	4	95W
Core i7 Gulftown	2010	3460MHz	16	4	**Yes**	6	130W

Streaming SIMD Extensions (SSE)

- SIMD: Single Instruction Multiple Data
- A data parallel architecture
- Both current AMD and Intel's x86 processors have ISA and micro-architecture support SIMD operations
  - MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX
  - Many functional units
  - 8 128-bit vector registers: XMM0, XMM1, …, XMM7
  - See the flag field in /proc/cpuinfo
- SSE (Streaming SIMD extensions): a SIMD instruction set extension to the x86 architecture
  - Instructions for operating on multiple data simultaneously (vector operations): for (i=0; i<n; ++i) Z[i]=X[i]+Y[i];
- Programming SSE in C++: **intrinsics**

Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Takeaway

- Pipelining is an important form of ILP
- Challenge is hazards
  - Forwarding helps with many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
  - Load delay slot / interlock necessary
- More aggressive performance:
  - Longer pipelines
  - VLIW
  - Superscalar
  - Out-of-order execution
  - Speculation

  - SSE?