Contents

History

- Started with (horizontal) microprogramming
  - Very wide microinstructions used to directly generate control signals in single-issue processors
- VLIW for multi-issue processors first appeared in the Multiflow and Cydrome mini supercomputers (in the early 1980s)
- Currently commercial VLIW processors
  - Intel IA-64
  - Transmeta Crusoe
  - Philips Trimedia VLIW (TV set-top boxes)
  - IBM Cell Broadband Engine (PS3)
  - Movidius Myriad 2 (drones, cameras)
  - Texas Instruments TI C6670 (digital signal processing)

Static Multiple Issue Machines

- Static multiple-issue processors (aka VLIW) use the compiler to 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
- VLIWs have
  - Multiple functional units (like superscalar processors)
  - Multi-ported register files (again like superscalar processors)
  - Wide program bus

A VLIW MIPS Processors

- Consider a 2-issue MIPS with a 2 instruction bundle
- Instructions are always fetched, decoded and issued in pairs
  - If one instruction of the pair can not be used, it is replaced with a `noop`
- Need 4 read ports and 2 write ports and a separate memory address adder

Code Scheduling

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
```

- 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:`	`lw $t0, 0 ($s1)`		1
	`addi $s1, $s1, -4`		2
	`addu $t0, $t0, $s2`		3
	`bne $s1, $0, lp`	`sw $t0, 4 ($s1)`	4

- 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)
  - `noop`s don't count towards performance!

Loop Unrolling

- Loop unrolling - multiple copies of the loop body are made and instructions from different interactions are scheduled together as a way to increase ILP
  - An alternative approach is software pipelining
- 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 flow) dependencies

```
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)

	ALU or Branch	Data Transfer	CC
`lp:`	`addi $s1,$s1,-16`	`lw $t0,0($s1)`	1
	`addi $s1, $s1, -4`	`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

- Each 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)

How to Schedule Code

- Very many scheduling algorithms exist
- Simplest is list scheduling
  - Build a dependence graph
    * Directed graph showing dependencies between instructions
  - Add latency of instructions to the edges
  - For each node, compute total latency to end of code segment
    * Often referred to as the "height" of the instruction
  - Any node without incoming edges is ready for scheduling
  - Schedule the nodes with the greatest height first - remove
- List scheduling is very successful
  - Most scheduling algorithms used in practice for VLIW processors are based on list scheduling
  - Simple fast algorithm
  - Produces good schedules most of the time

Example: Assume all instructions have a latency of 1

1. `num = p->num;`

```
2. num = num + scalar;
3. p->num = num;
4. p = p->next;
5. num = p->num;
6. num = num + scalar;
7. p->num = num;
8. p = p->next;
```

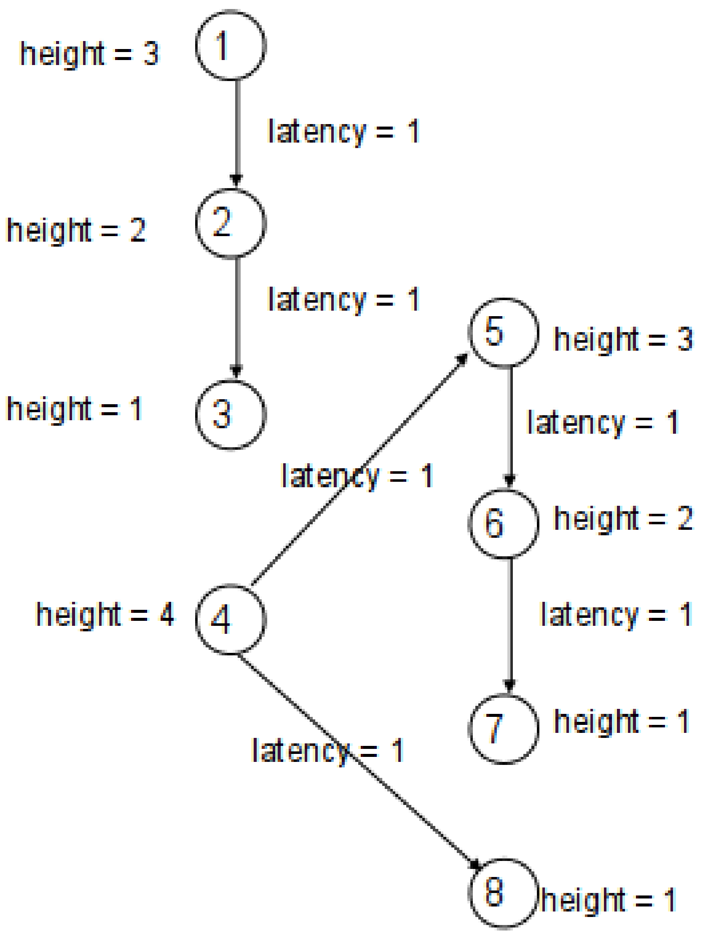

Figure 1: Dependence Graph

In this dependence graph, only two dependencies are shown. This means that we will have to eliminate false dependencies while scheduling (by renaming variables). It is also common to represent output (WAW) and anti (WAR) dependencies in the dependence graph.

Schedule on Two-Wide VLIW

Original Sequence Code

1. `num = p->num;`
2. `num = num + scalar;`
3. `p->num = num;`
4. `p = p->next;`
5. `num = p->num;`
6. `num = num + scalar;`
7. `p->num = num;`
8. `p = p->next;`

Actual Cycle:

Cycle		
1	`p p->next`	`num = p->num;`
2	`num = p->num`	`num = num + scalar`
3	`num = num + scalar`	`p->num = num;`
4	`p->num = num`	`p = p->next`

Speculation

- Speculation is used to allow execution of future instructions 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*)
- Must have (hardware and/or software) mechanisms for
  - Check to see if the guess was correct
  - Recording from the effects of the instructions that were executed speculatively if the guess was incorrect
    * In a VLIW processor the compiler can insert additional instructions that check the accuracy of the speculation and can provide a fix-up routine to use when the speculation was incorrect

- Ignore and/or buffer exceptions created by speculatively executed instructions unless it is clear that they should really occur

Predication

- Predication can be used to eliminate branches by making the execution of an instruction dependent on a "predicate", e.g. `if (p) { statement 1 } else { statement 2 }` would normally compile using two branches. With predication is would compile as

```
(p) statement 1

(~p) statement 2
```

- The use of `(condition)` indicates that the instruction is committed only if `condition` is true
- Predication can be used to speculate as well as to eliminate branches

Compiler Support for VLIW Processors

- The compiler packs groups of *independent* instructions into the bundle
  - Done by code re-ordering (list scheduling)
- The compiler uses loop unrolling to expose more ILP
  - Or other techniques such as software pipelining
- The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur
- While superscalars use dynamic prediction, VLIWs primarily depend on the compiler for branch prediction
  - Loop unrolling reduces the number of conditional branches
  - Prediction eliminates if-then-else branch structures by replacing them with predicated instructions
- VLIW processors need very good compilers
  - They need all the optimisations for regular RISC processors
  - And a good program analysis for pointer
  - And a good array dependence analysis
  - And a good compile-time prediction
  - And good instruction scheduling and register allocation

Advantages and Disadvantages (vs Superscalar)

- Advantages
  - Simpler hardware (potentially less power hungry)
    - Many processors aimed specifically at digital signal processing (DSP) are VLIW
      - DSP applications usually contain lots of instruction level parallelism in loops that execute many times
  - Potentially more scalable
    - Allow more instructions per VLIW bundle and add more functional units
- 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 incompatability*
  - Needs lots of program memory bandwidth
  - *Code bloat*
    - `nop`s are a waste of program memory space
    - Speculative code needs additional fix-up code to deal with cases where speculation has gone wrong
    - Register renaming needs free registers, so VLIWs usually have more programmer-visibile registers than other architectures
    - Loop unrolling (and software pipelining) to expose more ILP uses more program memory space