Module 3: "Recap: Single-threaded Execution"

Lecture 6: "Instruction Issue Algorithms"

The Lecture Contains:

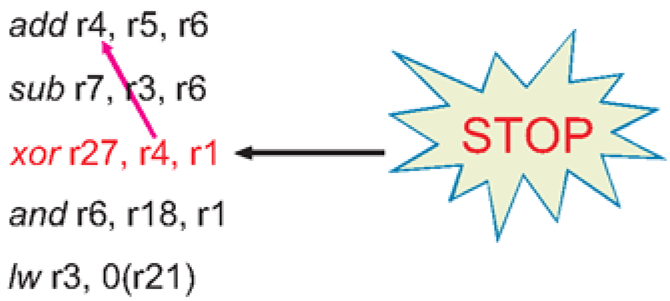
Previous Next

Module 3: "Recap: Single-threaded Execution"

Lecture 6: "Instruction Issue Algorithms"

Instruction selection

- Simplest possible design
  - Issue the instructions sequentially (in-order)
  - Scan the issue queue, stop as soon as you come to an instruction dependent on one already issued



- Cannot issue the last two even though they are independent of the first two: in-order completion is a must for precise exception support

In-order multi-issue

- Complexity of selection logic
  - Need to check for RAW and WAW
  - Comparisons for RAW: N(N-1) where N is the issue width
  - Comparisons for WAW: N(N-1)/2
  - 18 comparators for 4-issue
- Still need to make sure instructions write back in-order to support precise exception
  - As instructions issue, they are removed from the issue queue and put in a re-order buffer (also called active list in MIPS processors) [Isn't WAW check sufficient?]
  - Instructions write back or retire in-order from re-order buffer (ROB)

Out-of-order issue

- Taking the parallelism to a new dimension
- Central to all modern microprocessors
- Scan the issue queue completely, select independent instructions and issue as many as possible limited only by the number of functional units
- Need more comparators
- Able to extract more ILP: CPI goes down further
- Possible to overlap the latency of mult/div, load/store with execution of other independent instructions

lw r4, 0(r6)　　　Cache miss

addi r5, r4, 0x20

and r10, r5, r19

xor r26, r5, r7

sub r20, r26, r2

andi r27, r8, 0xffff

sll r19, r27, 0x5

beq r20, r19, label

or r12, r15, r16

- Issue first cycle, issue second cycle,...

Previous Next

WAR hazard

```
lw r4, 0(r6)          Cache miss
addi r5, r4, 0x20
and r10, r5, r19
xor r26, r5, r7
sub r20, r26, r2
andi r27, r8, 0xffff
sll r19, r27, 0x5
beq r20, r19, label
or r12, r15, r16
```

- Write After Read (WAR): in-order commit solves it

Modified bypass

- An executing instruction must broadcast results to the issue queue
  - Waiting instructions compare their source register numbers with the destination register number of the bypassed value
  - Also, now it needs to make sure that it is consuming the right value in program order to avoid WAR

```
add r19, r2, r3
sub r20, r19, r3
xori r19, r4, 0xf
and r22, r19, r1
```

- Need to tag every instruction with its last producer
- Can we simplify this?

WAR and WAW

- These are really false dependencies
  - Arises due to register allocation by the compiler
- Thus far we have assumed that ROB has space to hold the destination values: needs wide ROB entries
- These values are written back to the register file when the instructions retire or commit in-order from ROB
- Also, bypass becomes complicated
- Better way to solve it: rename the destination registers

Register renaming

- Registers visible to the compiler
  - Logical or architectural registers
  - Normally 32 in number for RISC and is fixed by the ISA
- Physical registers inside the processor
  - Much larger in number
- The destination logical register of every instruction is renamed to a physical register number
- The dependencies are tracked based on physical registers
- MIPS R10000 has 32 logical and 64 physical regs
- Intel Pentium 4 has 8 logical and 128 physical regs

- Assume 64 physical regs and already renamed registers:
  r6=p54, r19=p38, r2=p0, r7=p20, r15=p3, r16=p23

lw r4, 0(r6)	*lw* p15, 0(p54)	[r4 renamed to p15]
addi r5, r4, 0x20	*addi* p40, p15, 0x20	[r5 renamed to p40]
and r10,r5, r19	*and* p39, p40, p38	[r10 renamed to p39]
xor r26, r2, r7	*xor* p62, p0, p20	[r26 renamed to p62]
sub r20, r26, r2	*sub* p8, p62, p0	[r20 renamed to p8]
andi r27, r8, 0xffff	*andi* p19, p25, 0xffff	[r27 renamed to p19]
sll r19, r27, 0x5	*sll* p45, p19, 0x5	[r19 renamed to p45]
beq r20, r19, 0x5	*beq* p8, p45, label	
or r12, r15, r16	*or* p59, p3, p23	[r12 renamed to p59]

mult r5, r4, r3 [r5 gets renamed to, say, p50]

add r5, r6, r12 [r5 gets renamed to, say, p45]

- Now it is safe to issue them in parallel: they are really independent (compiler introduced WAW)
- Register renaming maintains a map table that records logical register to physical register map
- After an instruction is decoded, its logical register numbers are available
- The renamer looks up the map table to find mapping for the logical source regs of this instruction, assigns a free physical register to the destination logical reg, and records the new mapping
- If the renamer runs out of physical registers, the pipeline stalls until at least one register is available
- When do you free a physical register?
  - Suppose a physical register P is mapped to a logical register L which is the destination of instruction I
  - It is safe to free P only when the next producer of L retires (Why not earlier?)

- More physical registers
  - more in-flight instructions
  - possibility of more parallelism
- But cannot make the register file very big
  - Takes time to access
  - Burns power

Previous Next

The pipeline

- Fetch, decode, rename, issue, register file read, ALU, cache, retire
- Fetch, decode, rename are in-order stages, each handles multiple instructions every cycle
- The ROB entry is allocated in rename stage
- Issue, register file, ALU, cache are out-of-order
- Retire is again in-order, but multiple instructions may retire each cycle: need to free the resources and drain the pipeline quickly

What limits ILP now?

- Instruction cache miss (normally not a big issue)
- Branch misprediction
  - Observe that you predict a branch in decode, and the branch executes in ALU
  - There are four pipeline stages before you know outcome
  - Misprediction amounts to loss of at least 4F instructions where F is the fetch width
- Data cache miss
  - Assuming a issue width of 4, frequency of 3 GHz, memory latency of 120 ns, you need to find 1440 independent instructions to issue so that you can hide the memory latency: this is impossible (resource shortage)

Cycle time reduction

- Execution time = CPI × instruction count × cycle time
- Talked about CPI reduction or improvement in IPC (instructions retired per cycle)
- Cycle time reduction is another technique to boost performance
  - Faster clock frequency
- Pipelining poses a problem
  - Each pipeline stage should be one cycle for balanced progress
  - Smaller cycle time means need to break pipe stages into smaller stages
- Superpipelining
  - Faster clock frequency necessarily means deep pipes
  - Each pipe stage contains small amount of logic so that it fits in small cycle time
  - May severely degrade CPI if not careful
  - Now branch penalty is even bigger (31 cycles for Intel Prescott): branch mispredictions cause massive loss in performance (93 micro-ops are lost, F=3)
  - Long pipes also put more pressure on resources such as ROB and registers because instruction latency increases (in terms of cycles, not in absolute terms)
  - Instructions occupy ROB entries and registers longer
  - The design becomes increasingly complicated (long wires)

Previous Next

Alternative: VLIW

- Very Long Instruction Word computers
  - Compiler carries out all dependence analysis
  - Bundles as many independent instructions as allowed by the number of functional units into an instruction packet
  - Hardware is a lot less complex
  - The instructions in the packet issue in parallel
  - Each packet of instructions is pretty long (hence the name)
  - Problem: compiler may not be able to extract as much ILP as a dynamic out-of-order core; many packets may go unutilized
- Big leap from VLIW: EPIC (Explicitly Parallel Instruction Computing) [Itanium family]

Current research in µP

- Micro-architectural techniques to extract more ILP
  - Directly helps improve IPC and reduce CPI
  - Various speculative techniques to hide cache miss latency: prefetching, load value prediction, etc.
- Better branch prediction
  - Helps deep pipelines
- Faster clocking
  - Need to cool the chip
  - Various techniques to reduce power consumption: clock gating, dynamic voltage/frequency scaling (DVFS), power-aware resource usage
  - Fighting the long wires: scaling micro-architectures against the complexity wall

Previous Next