CS 6290
Instruction Level Parallelism
Instruction Level Parallelism (ILP)

• Basic idea: Execute several instructions in parallel
• We already do pipelining...
  – But it can only push through at most 1 inst/cycle
• We want multiple instr/cycle
  – Yes, it gets a bit complicated
    • More transistors/logic
  – That’s how we got from 486 (pipelined) to Pentium and beyond
Is this Legal?!?

- ISA defines instruction execution one by one
  - I1: ADD R1 = R2 + R3
    - fetch the instruction
    - read R2 and R3
    - do the addition
    - write R1
    - increment PC
  - Now repeat for I2

- Darth Sidious: Begin landing your troops.
- Nute Gunray: Ah, my lord, is that... legal?
- Darth Sidious: I will make it legal.
It’s legal if we don’t get caught...

• How about pipelining?
  – already breaks the “rules”
    • we fetch I2 before I1 has finished

• Parallelism exists in that we perform different operations (fetch, decode, ...) on several different instructions in parallel
  – as mentioned, limit of 1 IPC
Define “not get caught”

• Program executes correctly
• Ok, what’s “correct”?
  – As defined by the ISA
  – Same processor state (registers, PC, memory) as if you had executed one-at-a-time
    • You can squash instructions that don’t correspond to the “correct” execution (ex. misfetched instructions following a taken branch, instructions after a page fault)
Example: Toll Booth

Caravanning on a trip, must stay in order to prevent losing anyone

When we get to the toll, everyone gets in the same lane to stay in order

This works… but it’s slow. Everyone has to wait for D to get through the toll booth

Go through two at a time (in parallel)
Illusion of Sequentiality

• So long as everything looks OK to the outside world you can do whatever you want!
  – “Outside Appearance” = “Architecture” (ISA)
  – “Whatever you want” = “Microarchitecture”

  – μArch basically includes everything not explicitly defined in the ISA
    • pipelining, caches, branch prediction, etc.
Back to ILP... But how?

• Simple ILP recipe
  – Read and decode a few instructions each cycle
    • can’t execute > 1 IPC if we’re not fetching > 1 IPC
  – If instructions are independent, do them at the same time
  – If not, do them one at a time
Example

• A: ADD R1 = R2 + R3
• B: SUB R4 = R1 – R5
• C: XOR R6 = R7 ^ R8
• D: Store R6 → 0[R4]
• E: MUL R3 = R5 * R9
• F: ADD R7 = R1 + R6
• G: SHL R8 = R7 << R4
Ex. Original Pentium

- Fetch
  - Fetch up to 32 bytes
  - Decode up to 2 insts
  - Decode2
    - Decode2
      - Decode2
      - Execute
      - Writeback
    - Decode2
      - Execute
      - Writeback

Read operands and Check dependencies
Repeat Example for Pentium-like CPU

• A: ADD R1 = R2 + R3
• B: SUB R4 = R1 – R5
• C: XOR R6 = R7 ^ R8
• D: Store R6 \rightarrow 0[R4]
• E: MUL R3 = R5 * R9
• F: ADD R7 = R1 + R6
• G: SHL R8 = R7 << R4
This is “Superscalar”

• “Scalar” CPU executes one inst at a time
  – includes pipelined processors
• “Vector” CPU executes one inst at a time, but on vector data
  – $X[0:7] + Y[0:7]$ is one instruction, whereas on a scalar processor, you would need eight
• “Superscalar” can execute more than one unrelated instruction at a time
  – ADD $X + Y$, MUL $W * Z$
Scheduling

• Central problem to ILP processing
  – need to determine when parallelism (independent instructions) exists
  – in Pentium example, decode stage checks for multiple conditions:
    • is there a data dependency?
      – does one instruction generate a value needed by the other?
      – do both instructions write to the same register?
    • is there a structural dependency?
      – most CPUs only have one divider, so two divides cannot execute at the same time
Scheduling

• How many instructions are we looking for?
  
  – 3-6 is typical today

  – A CPU that can ideally* do N instrs per cycle is called “N-way superscalar”, “N-issue superscalar”, or simply “N-way”, “N-issue” or “N-wide”

  • *Peak execution bandwidth

  • This “N” is also called the “issue width”
Dependences/Dependencies

• Data Dependencies
  – RAW: Read-After-Write (True Dependence)
  – WAR: Anti-Depedence
  – WAW: Output Dependence

• Control Dependence
  – When following instructions depend on the outcome of a previous branch/jump
Data Dependencies

• Register dependencies
  – RAW, WAR, WAW, based on register number

• Memory dependencies
  – Based on memory address
  – This is harder
    • Register names known at decode
    • Memory addresses not known until execute
Hazards

• When two instructions that have one or more dependences between them occur close enough that changing the instruction order will change the outcome of the program

• Not all dependencies lead to hazards!
ILP

• Arrange instructions based on dependencies
• ILP = Number of instructions / Longest Path

I1: R2 = 17
I2: R1 = 49
I3: R3 = -8
I4: R5 = LOAD 0[R3]
I5: R4 = R1 + R2
I6: R7 = R4 – R3
I7: R6 = R4 * R5
Dynamic (Out-of-Order) Scheduling

• Cycle 1
  - Operands ready? I1, I5.
  - Start I1, I5.

• Cycle 2
  - Start I2, I3.

• Window size (W):
  how many instructions ahead do we look.
  - Do not confuse with “issue width” (N).
  - E.g. a 4-issue out-of-order processor can have a 128-entry window (it can look at up to 128 instructions at a time).

Program code

\[
\begin{align*}
I1: & \text{ ADD R1, R2, R3} \\
I2: & \text{ SUB R4, R1, R5} \\
I3: & \text{ AND R6, R1, R7} \\
I4: & \text{ OR R8, R2, R6} \\
I5: & \text{ XOR R10, R2, R11} \\
\end{align*}
\]
Ordering?

• In previous example, I5 executed before I2, I3 and I4!
• How to maintain the illusion of sequentiality?

Hands toll-booth agent a $100 bill; takes a while to count the change

With a “4-Issue” Toll Booth

One-at-a-time = 45s

OOO = 30s
ILP ≠ IPC

• ILP is an attribute of the program
  – also dependent on the ISA, compiler
    • ex. SIMD, FMAC, etc. can change inst count and shape of dataflow graph

• IPC depends on the actual machine implementation
  – ILP is an upper bound on IPC
    • achievable IPC depends on instruction latencies, cache hit rates, branch prediction rates, structural conflicts, instruction window size, etc., etc., etc.

• Next several lectures will be about how to build a processor to exploit ILP
CS 6290
Dependences and Register Renaming
ILP is Bounded

• For any sequence of instructions, the available parallelism is limited

• Hazards/Dependencies are what limit the ILP
  
  – Data dependencies
  – Control dependencies
  – Memory dependencies
Types of Data Dependencies

(Assume A comes before B in program order)

• RAW (Read-After-Write)
  – A writes to a location, B reads from the location, therefore B has a RAW dependency on A
  – Also called a “true dependency”
Data Dep’s (cont’d)

- WAR (Write-After-Read)
  - A reads from a location, B writes to the location, therefore B has a WAR dependency on A
  - If B executes before A has read its operand, then the operand will be lost
  - Also called an anti-dependence
Data Dep’s (cont’d)

• Write-After-Write
  – A writes to a location, B writes to the same location
  – If B writes first, then A writes, the location will end up with the wrong value
  – Also called an output-dependence
Control Dependencies

• If we have a conditional branch, until we actually know the outcome, *all* later instructions must wait
  – That is, all instructions are control dependent on all earlier branches
  – This is true for unconditional branches as well (e.g., can’t return from a function until we’ve loaded the return address)
Memory Dependencies

• Basically similar to regular (register) data dependencies: RAW, WAR, WAW

• However, the exact location is not known:
  – A: STORE R1, 0[R2]
  – B: LOAD R5, 24[R8]
  – C: STORE R3, -8[R9]

  – RAW exists if (R2+0) == (R8+24)
  – WAR exists if (R8+24) == (R9 – 8)
  – WAW exists if (R2+0) == (R9 – 8)
Impact of Ignoring Dependencies

### Read-After-Write

A: $R_1 = R_2 + R_3$
B: $R_4 = R_1 \times R_4$

### Write-After-Read

A: $R_1 = R_3 / R_4$
B: $R_3 = R_2 \times R_4$

### Write-After-Write

A: $R_1 = R_2 + R_3$
B: $R_1 = R_3 \times R_4$
Eliminating WAR Dependencies

- WAR dependencies are from reusing registers

\[
\begin{align*}
A: & \quad R1 = R3 / R4 \\
B: & \quad R3 = R2 * R4
\end{align*}
\]

With no dependencies, reordering still produces the correct results.
Eliminating WAW Dependencies

- WAW dependencies are also from reusing registers

A: $R1 = R2 + R3$
B: $R1 = R3 \times R4$

A: $R5 = R2 + R3$
B: $R1 = R3 \times R4$

Same solution works
So Why Do False Dep’s Exist?

• Finite number of registers
  – At some point, you’re forced to overwrite somewhere
  – Most RISC: 32 registers, x86: only 8, x86-64: 16
  – Hence WAR and WAW also called “name dependencies” (i.e. the “names” of the registers)

• So why not just add more registers?

• Thought exercise: what if you had infinite regs?
Reuse is Inevitable

• Loops, Code Reuse
  – If you write a value to R1 in a loop body, then R1 will be reused every iteration → induces many false dep’s
    • Loop unrolling can help a little
      – Will run out of registers at some point anyway
      – Trade off with code bloat
  – Function calls result in similar register reuse
    • If printf writes to R1, then every call will result in a reuse of R1
    • Inlining can help a little for short functions
      – Same caveats
Obvious Solution: More Registers

- Add more registers to the ISA? **BAD!!!**
  - Changing the ISA can break binary compatibility
  - All code must be recompiled
  - Does not address register overwriting due to code reuse from loops and function calls
  - Not a scalable solution

**BAD?** x86-64 adds registers…
… but it does so in a mostly backwards compatible fashion
Better Solution: HW Register Renaming

• Give processor more registers than specified by the ISA
  – temporarily map ISA registers (“logical” or “architected” registers) to the physical registers to avoid overwrites

• Components:
  – mapping mechanism
  – physical registers
    • allocated vs. free registers
    • allocation/deallocation mechanism
Register Renaming

- Example
  - I3 can not exec before I2 because I3 will overwrite R6
  - I5 can not go before I2 because I2, when it goes, will overwrite R2 with a stale value

<table>
<thead>
<tr>
<th>Program code</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1: ADD R1, R2, R3</td>
</tr>
<tr>
<td>I2: SUB R2, R1, R6</td>
</tr>
<tr>
<td>I3: AND R6, R11, R7</td>
</tr>
<tr>
<td>I4: OR R8, R5, R2</td>
</tr>
<tr>
<td>I5: XOR R2, R4, R11</td>
</tr>
</tbody>
</table>

RAW →
WAR →
WAW →
Register Renaming

• Solution:
  Let’s give I2 temporary name/location (e.g., S) for the value it produces.

• But I4 uses that value, so we must also change that to S...

• In fact, all uses of R5 from I3 to the next instruction that writes to R5 again must now be changed to S!

• We remove WAW deps in the same way: change R2 in I5 (and subsequent instrs) to T.
Register Renaming

• Implementation
  – Space for S, T, U etc.
  – How do we know when to rename a register?

• Simple Solution
  – Do renaming for every instruction
  – Change the name of a register each time we decode an instruction that will write to it.
  – Remember what name we gave it 😊

Program code

I1: ADD R1, R2, R3
I2: SUB S, R1, R5
I3: AND U, R11, R7
I4: OR R8, R5, S
I5: XOR T, R4, R11
Register File Organization

- We need some physical structure to store the register values.

- "Outside" world sees the ARF

- One PREG per instruction in-flight
Putting it all Together

top:

• R1 = R2 + R3
• R2 = R4 − R1
• R1 = R3 * R6
• R2 = R1 + R2
• R3 = R1 >> 1
• BNEZ R3, top
Renaming in action

```plaintext
R1 = R2 + R3  X 9 = R2 + R3
R2 = R4 - R1  X 11 = R4 - X 9
R1 = R3 * R6  X 7 = R3 * R6
R2 = R1 + R2  X 2 = X 7 + X 11
R3 = R1 >> 1  X 13 = X 7 >> 1
BNEZ R3, top BNEZ X 13, top
R1 = R2 + R3  X 5 = X 2 + X 13
R2 = R4 - R1  X 8 = R 5 - X 9
R1 = R3 * R6  X 17 = X 13 * R6
R2 = R1 + R2  X 3 = X 17 + X 9
R3 = R1 >> 1  X 5 = X 17 >> 1
BNEZ R3, top BNEZ X 5, top
```
Even Physical Registers are Limited

• We keep using new physical registers
  – What happens when we run out?
• There must be a way to “recycle”

• When can we recycle?
  – When we have given its value to all instructions that use it as a source operand!
  – This is not as easy as it sounds
Architected register file contains the “official” processor state

When an instruction leaves the pipeline, it makes its result “official” by updating the ARF

The ARF now contains the correct value; update the RAT

T42 is no longer needed, return to the physical register free pool
Careful with the RAT Update!

Update ARF as usual
Deallocate physical register
Don’t touch that RAT!
(Someone else is the most recent writer to R3)

At some point in the future, the newer writer of R3 exits

This instruction was the most recent writer, *now* update the RAT
Deallocate physical register
Instruction Commit: a Problem

I1: ADD R3, R2, R1
I2: ADD R7, R3, R5
I3: ADD R6, R1, R1

Decode I1 (rename R3 to T42)
Decode I2 (uses T42 instead of R3)
Execute I1 (Write result to T42)
I2 can’t execute (e.g. R5 not ready)
Commit I1 (T42->R3, free T42)
Decode I3 (uses T42 instead of R6)
Execute I3 (writes result to T42)
R5 finally becomes ready
Execute I2 (read from T42)
We read the wrong value!!

Think about it!