Until today, we assumed that instructions execute sequentially one after another in a von Neumann style of execution. However, today's processors do not execute instructions in this model; architectural "enhancements" allow processors to execute code faster:

- pipelining
- multiple function units

Instruction scheduling refers to re-ordering instructions in a program to exploit Instruction Level Parallelism (ILP) and improve performance.

Instruction scheduling is still an active area of research because of the difficulty of the problem (NP-complete) and the changing natures of processors.

In the von Neumann model of execution, an instruction starts only after its predecessor completes. This is not a very efficient model of execution.

Almost all processors today use instruction pipelines to allow overlap of instructions (Pentium 4 has a 20 stage pipeline!!!).

The execution of an instruction is divided into stages; each stage is performed by a separate part of the processor.

Instruction Pipelines

- Almost all processors today use instruction pipelines to allow overlap of instructions (Pentium 4 has a 20 stage pipeline!!!).
- The execution of an instruction is divided into stages; each stage is performed by a separate part of the processor.

Instruction Scheduling

- In the von Neumann model of execution, an instruction starts only after its predecessor completes.
- This is not a very efficient model of execution.
  - von Neumann bottleneck or the memory wall.

Instruction Pipelines

- However, we overlap these stages in time to complete an instruction every cycle.

Instruction Hazards

- Structural Hazards
  - two instructions need the same resource at the same time
  - memory or functional units in a superscalar.
- Data Hazards
  - an instructions needs the results of a previous instruction
  - solved by forwarding and/or stalling
  - cache miss?
- Control Hazards
  - jump & branch address not known until later in pipeline
  - solved by delay slot and/or prediction
Jump/Branch Delay Slot(s)

- Control hazards, i.e. jump/branch instructions.
  - unconditional jump address available only after Decode.
  - conditional branch address available only after Execute.

Jump/Branch Delay Slot(s)

- One option is to stall the pipeline (hardware solution).
  - jump FDEMW
  - instr 2 FDEMW

Jump/Branch Delay Slot(s)

- Another option is to insert a no-op instructions (software).
  - jump FDEMW
  - nop FDEMW
  - instr 2 FDEMW

Both degrade performance!

Jump/Branch Delay Slot(s)

- A better solution is to make the branch take effect only after the delay slots.
  - That is, one or two instructions always get executed after the branch but before the branching takes effect.

Jump/Branch Delay Slot(s)

- In other words, the instruction(s) in the delay slots of the jump/branch instruction always get(s) executed when the branch is executed (regardless of the branch result).
  - Fetching from the branch target begins only after these instructions complete.

What instruction(s) to use?

Branch Prediction

- Current processors will speculatively execute at conditional branches.
  - if a branch direction is correctly guessed, great!
  - if not, the pipeline is flushed before instructions commit (WB).

- Why not just let compiler schedule?
  - The average number of instructions per basic block in typical C code is about 5 instructions.
  - branches are not statically predictable
  - What happens if you have a 20 stage pipeline?

Data Hazards

- r1 = r2 + r3
- r4 = r1 + r1
- r1 = [r2]
- r4 = r1 + r1
- [r3] available here
- [r3] available here
- r1 = r2 + r3
- r4 = r1 + r1
- FDEMW
- [r3] available here
To schedule a basic block, you need to determine scheduling constraints and express these using a dependence graph. For a basic block, this graph is a DAG. Each node is a machine instruction and the edges are the dependencies between instructions.

A flow dependence exists if an instruction \( I_1 \) writes to a register or location that \( I_2 \) uses. This is written \( I_1 \delta^f I_2 \). Flow dependencies are true dependencies, that is these dependencies are necessary to transmit information between statements.

An anti dependence exists if an instruction \( I_1 \) uses a register that \( I_2 \) changes. This is written \( I_1 \delta^a I_2 \). Anti dependencies are false dependencies, that is they arise due to the reuse of memory locations.

An output dependence exists if an instruction \( I_1 \) writes to a register that \( I_2 \) also writes to. This is written \( I_1 \delta^o I_2 \). Output dependencies are also false dependencies, that is they arise due to the reuse of memory locations.

List Scheduling Algorithm - Example

1. load R1, b
2. load R2, c
3. add R2, R1
4. store a, R2
5. load R3, e
6. load R4, f
7. sub R3, R4
8. store d, R3

Assume, that loaded values are available after 2 cycles (from beginning of load instruction). So, there is need for an extra cycle after 2 and after 6. In the absence of instruction scheduling, NOPs must be inserted.

Step 1: construct a dependence graph of the basic block. (The edges are weighted with the latency of the instruction).
Step 2: use the dependence graph to determine instructions that can execute; insert on a list, called the Ready list.
Step 3: use the dependence graph and the Ready list to schedule an instruction that causes the smallest possible stall; update the Ready list. Repeat until Ready list is empty!
List Scheduling Algorithm - Example

1. load R1, b
2. load R2, c
3. add R2, R1
4. store a, R2
5. load R3, e
6. load R4, f
7. sub R3, R4
8. store d, R3

\[ a = b + c \]
\[ d = e - f \]

We're done. Now have a schedule that requires no stalls and no NOPs.

Superscalars, i.e. multiple functional units

- Almost all modern processors are superscalars
  - have multiple functional units
  - Intel 486: 1 pipeline; Pentium: 2 pipelines; Pentium 4: up to 6 instructions per clock cycle.
- Need to model the CPU as accurately as possible
  - which instructions can execute simultaneously
  - relative delay of different types of instructions
- Can use a Greedy / Ready list method
  - not always optimal, nontrivial scheduling is NP

Trace Scheduling

- Basic blocks typically contain a small number of instructions.
- With many function units, we may not be able to keep all the units busy with just the instructions of a basic block.
- Trace scheduling allows block scheduling across basic blocks.
- The basic idea is to dynamically determine which blocks are executed more frequently. The set of such basic blocks is called a trace.

Trace Scheduling

- Blocks that are not part of the trace must be modified to restore program semantics if/when execution goes off-trace.
Instruction Scheduling for Loops

- Loop bodies are typically too small to produce a schedule that exploits all resources.
- But, most of execution is spent in loops.
- Need ways to schedule loops:
  - Loop unrolling.
  - Software pipelining.
- Focus on the main ideas; the details are considerable.

Loop Example

- Machine parameters:
  - 1 memory unit capable of either a load or a store. Each operation takes 2 cycles. No delay slots.
  - One multiplier unit. A multiply operation takes 3 cycles.
  - One adder unit. An add operation takes 2 cycles.
  - The adder and the multiplier are capable of performing a branch operation in 2 cycles.
  - All units are pipelined, allowing the initiation of an operation per one clock cycle.

Loop:

```plaintext
for i = 1 to N
  a[i] = a[i]*b
L1:
  r6 = *(r2) (ld)
  r6 = r6*r3 (mul)
  *(r2) = r6 (st)
  r2 = r2 + 4 (add)
  if (r2 <= r5) go to L1 (ble)
```

Multiple issue.
Add takes 2 cycles; latency = 1
Multiply takes 3 cycles; latency = 2

```
mul R2,R0,R1
add R3,R3,R2
add R4,R0,R1
add R5,R5,R4
```

```
MUL
ADD
```

Block Scheduling

```
L1:
  r6 = *(r2) (ld)
  r6 = r6*r3 (mul)
  r2 = r2 + 4 (add)
  if (r2 <= r5) go to L1 (ble)
```

```
load R6, (R1)
mul R6,R6,R3
store (R1), R6
add R1,R1,#8
```

```
load R6, (R1)
mul R6,R6,R3
store (R1), R6
add R1,R1,#4
cmp R1,R5
turn Loop
ble Loop
```
Loop Unrolling

L1:
- \( r_6 = \times'(r_2) \) (ld)
- \( r_6 = r_6 \times r_3 \) (mul)
- \( r_2 = r_2 + 4 \) (add)
- \( r_6 = \times'(r_2) \) (ld)
- \( r_6 = r_6 \times r_3 \) (mul)
- \( r_2 = r_2 + 4 \) (add)
- if \( r_2 <= r_5 \) go to L1 (ble)

Register Re-naming

L1:
- \( r_6 = \times'(r_1) \) (ld)
- \( r_6 = r_6 \times r_3 \) (mul)
- \( r_1 = r_1 + 8 \) (add)
- if \( r_1 <= r_5 \) go to L1 (ble)

Software Pipelining

- Software pipelining: overlap multiple iterations of a loop to fully utilize hardware resources.

Find the steady-state window so that:
- all the instructions of the loop body is executed
- but from different iterations

<table>
<thead>
<tr>
<th>( r_2 )</th>
<th>( r_5 )</th>
<th>( r_2 )</th>
<th>( r_5 )</th>
<th>( r_2 )</th>
<th>( r_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>rd</td>
<td>st</td>
<td>st</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>id</td>
<td>id</td>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td>mul</td>
<td>mul</td>
<td>bie</td>
<td>mul</td>
<td>bie</td>
<td>mul</td>
</tr>
<tr>
<td>mul</td>
<td>mul</td>
<td>bie</td>
<td>mul</td>
<td>bie</td>
<td>mul</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td>bie</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td>bie</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
</tbody>
</table>

Software Pipelining

- SOFTWARE PIPELINED ITERATION

Iteration 1
- Iteration 2
- Iteration 3
- Iteration 4
- Iteration 5

Software Pipelining

PRELUDE
Software Pipelining: An Example

**Iteration 1**
- LD F0, 0(R1)
- ADD D F4, F0, F2
- SD 0(R1), F4
- SUB R1, R1, #8
- BNEZ R1, Loop

**Iteration 2**
- LD F0, 0(R1)
- ADD D F4, F0, F2
- SD 0(R1), F4
- SUB R1, R1, #8
- BNEZ R1, Loop

**Iteration 3**
- LD F0, 0(R1)
- ADD D F4, F0, F2
- SD 0(R1), F4
- SUB R1, R1, #8
- BNEZ R1, Loop

**Iteration 4**
- LD F0, 0(R1)
- ADD D F4, F0, F2
- SD 0(R1), F4
- SUB R1, R1, #8
- BNEZ R1, Loop

**Iteration 5**
- LD F0, 0(R1)
- ADD D F4, F0, F2
- SD 0(R1), F4
- SUB R1, R1, #8
- BNEZ R1, Loop

---

Loop Unrolling and Software Pipelining

- **Loop Unrolling**
  - Helps uncover Instruction Level Parallelism (ILP)
  - Reduces looping overhead (increment and branch)
  - Generates a lot of code, copies of the loop body

- **Software Pipelining**
  - Also helps uncover ILP
  - Does not reduce looping overhead
  - Loop body is always executing at top speed
  - Usually uses less code space

Both require that number of iterations is known

If unroll factor does not evenly divide iterations, the extra iterations must be caught by a pre- or post-amble
Emerging Architectures

- Simultaneous Multithreading (SMT)
  - Execute multiple threads simultaneously
  - Keeps functional units busy
  - Example: Intel Pentium IV with HyperThreading (HT)
  - Sun: 2 cores both with SMT (an SMT CMP)
- EPIC / VLIW Architectures
  - EPIC – Explicitly Parallel Instruction Computing (Itanium / IA64)
  - VLIW – Very Long Instruction Word (Transmeta)
  - Compiler explicitly packages independent instructions
  - Hardware does not due reordering
  - Chip can run at higher clock rates
  - Some may live, while some may die
  - Which one will compilers work best with?

EPIC / VLIW

- Finding independent instructions at runtime takes time.
- Less logic means faster clock cycle!?!?
- Can the compiler explicitly group independent instructions?
- VLIW: Very Long Instruction Word
  - schedule for a particular microarchitecture
  - timing may be important, number of functional units is fixed
- EPIC: Explicitly Parallel Instruction Computing
  - number of functional units is not fixed, hardware interlocks
  - speculation support
  - predication support

VLIW -vs- Superscalar

- VLIW
  - use very long multi-operation instructions
  - the instruction specifies what each functional unit is to do
  - expects dependence free instructions
  - compiler must explicitly detect and schedule independent instructions
- Superscalar
  - uses traditional sequential operations
  - processor fetches multiple instructions per cycle
  - detects dependencies and schedules accordingly
  - has dynamic information available
  - compiler can help by placing independent operations close to each other

Problems with VLIW

- Compiler must statically determine dependencies
- Compiler must have very detailed model of architecture
  - number and type of functional units
  - delays for each operation
  - memory delays
  - latencies are very important
- A new generation with more units or different latencies means recompile
- EPIC (Itanium / IA64) tries to address some of these
  - compiler expresses parallelism; hardware schedules ops
  - no fixed length to instructions, just a number of bundles
  - relationship of one bundle to another is expressed

Predication: Branches are Bad

- Provide predicate registers and predicated instructions
- Can set a predicate register to true or false using a comparison instruction
- Most instructions can be predicated so that they only commit if their predicate is true
- Can schedule and execute across multiple directions of a branch and only valid instructions will commit

Speculation

- Control Speculation
  - move an instruction above a branch instruction
  - may be done if it is always safe
  - or can be done speculatively
  - br.cond label
  - chk.s label
  - load stores speculatively loads a value, if the load fails a NAT bit is set
  - NAT bits propagate through all other uses
  - the chk.s instructions checks the NAT bit, if set it calls the fixupcode
Speculation

- Data Speculation
  - move a load before a store that may be aliased
  - may be done if it is always safe
  - or can be done speculatively

ld8 r4 = [r1]
ld8 r5 = [r2]
ld8 a r6 = [r3]
add ret0 = r6, -1
br.ret

ld8 r4 = [r1]
ld8 r5 = [r2]
ld8 a r6 = [r3]
add ret0 = r6, -1
br.ret

do the Advanced Load Address Table (ALAT) checks for collisions

EPIC / Itanium Processor Family

- Prediction and Speculation
- Special support for software pipelining
  - rotating registers
  - special epilogue counters
- Performance????
  - need more compiler work
  - a big change and it's still early
  - does ok on floating-point
  - profiling may help
  - runtime techniques may help
- What are the issues?

We’ve now covered optimizations found in most commercial compilers

- Compilers improve performance dramatically
- The optimizations in this class improve code
- A little test: compile gcc using cc

CC Optimization Levels

-xO1 Does basic local optimization (peephole).

-xO2 Does basic local and global optimization.
This is induction variable elimination, local and
global common subexpression elimination,
algebraic simplification, copy propagation,
constant propagation, loop-invariant optimization,
register allocation, basic block merging, tail
recursion elimination, dead code elimination,
tail call elimination and complex expression expansion.

CC Optimization Levels

-xO3 Performs like -xO2 but, also optimizes references
or definitions for external variables. Loop unrolling
and software pipelining are also performed. In
general, the -xO3 level results in increased code size.
Does not deal with pointer disambiguation.

-xO4 Performs like -xO3 but, also does automatic inlining
of functions contained in the same file; this usually
improves execution speed. The -xO4 level does trace
the effects of pointer assignments. In general, the
-xO4 level results in increased code size.

CC Optimization Levels

-xO5 Generates the highest level of optimization.
Uses optimization algorithms that take more
compilation time or that do not have as high a
certainty of improving execution time.

-fast: -O4 plus some other specific flags…
Performance of CC Opt Levels

<table>
<thead>
<tr>
<th>Option</th>
<th>Ctime</th>
<th>Rtime</th>
<th>% Imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>-xO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-xO1</td>
<td>41%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-xO2</td>
<td>20%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-xO3</td>
<td>11%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GCC from SPEC 2000

% Imp: -xO0 → -xO1 = 41%; -xO1 → -xO2 = 20%; -xO2 → -xO3 = 11%