Some loose topics
(Before getting started on memory systems)

- Branch Predictors
Without branch prediction

1. Is inst1 a branch instruction?
2. If so, will the branch will be taken?
3. If so, where is the branch target?
Without these answers, we can’t know for sure what to fetch!

Too much performance loss!
Branch predictor predicts what should be the next PC
- Typically based on the current PC as input

Dynamic branch predictors adapt to program using feedback

If prediction is correct, great! If not, make sure mispredicted instructions don’t effect state
- We looked at the epoch method of doing this (2 bubbles!)
Dynamic branch prediction

- Two questions about a PC address being fetched
  - Will this instruction cause a branch?
  - If so, where will it branch to?
  - Both information is needed to predict-fetch a branch

- Two architectural entities for predicting the answer to these questions
  - Branch History Table (BHT)
    - Whether this instruction is an instruction, and if it causes a branch
  - Branch Target Buffer (BTB)
    - Which address this instruction will jump to

(There are many variations – This is just a common example)
Dynamic branch prediction

```
method Word predict(Word pc) begin
    Word next_pc = pc + 4;
    Bit#(10) lsb = truncate(pc);
    if ( bht[lsb] ) next_pc = btb[lsb];
    return next_pc;
end
```

Why truncate PC? BHT/BTB is typically small! (2048 elements or so)
Different branches may map to same buffer element... 😐

PC + 4 prediction

Execute stage updates BHT and BTB with actual behavior (if it is a branch instruction)
Back to the three questions

- Is it a branch instruction?
  - Execute updates BHT if it is a branch instruction

- Is the branch taken?
  - BHT stores if the branch was taken last time

- Where does the branch go?
  - BTB stores where it went to last time

- Of course, all three are merely predictions!
Simple example: 1-bit predictor

- BHT has one-bit entries
  - Most recently taken/not taken
  - ("Last time predictor")
  - Does this work well?

- How many mispredicts with these taken (T), not taken (N) sequences?
  - TTTTTTNNNNN  TTTTNNNNN
  - TNTNTNTNTN  TNTNTNTNTN
  - for (i = 0 ... 2) {
    for (j = 0 ... 2 ) {
    }
  } Mispredict at j = 0 (T), j = 2 (N)
Simple example: 2-bit predictor

- BHT has two bits – Single outlier does not change future predictions
  1. 00: Strongly not taken, 01: Not taken, 10: Taken, 11: Strongly taken
  2. Taken branch increases number, not taken branch decreases number
  3. Counter saturates! Taken after 11 -> 11, Not taken after 00 -> 00

- How many mispredicts with these taken (T), not taken (N) sequences?
  1. TTTTTNNNNN TTTTTNNNN
  2. TNTNTNTNTN Initialized to 01: TNTNTNTNTN
  3. for (i = 0 ... 2) {
     for (j = 0 ... 2) {
     }
   } Mispredict once at i = 0 && j = 0 (T), j = 2 (N)

In reality, most SPEC benchmarks record ~90% accuracy with 2-bit predictor
Less simple example: Two-level adaptive predictor

- One of the most popular predictors in the real world
- Stores small amount of recent branch history per instruction
  - Branch history table now holds $N$ bits representing most recent branch history (e.g., “01” if $N = 2$, and recent history for that instruction is NT)
- Second layer of branch history tables, one of $2^N$ tables used according to first layer BHT value
  - If $N = 2$, four $2^{nd}$ layer branch history tables.
  - e.g., if first layer BHT returned “01”, use second level BHT1 to check branch history
  - Branch history may be the two-bit predictor from before
Less simple example: Two-level adaptive predictor

Note: Branch Target Buffer part not depicted!

>96% accuracy reported with SPEC benchmarks
Less simple example:
Two-level adaptive predictor

- Tracks branch behavior separately per history
  - Say, we track last 4 branches
  - Say, executing code “for ( int i = 0; i < 4; i++ ) A();”
    - Branch taken: Execute A again
  - After executing the loop once, the predictor is trained for this branch instruction:
    - 1110 -> Predict taken
    - 1101 -> Predict taken
    - 0111 -> Predict not taken, ...

- Branch history actually quite long
  - If loop length > history, then local history doesn’t help much!
  - But, long history means too many level 2 tables!
Less simple example: Two-level adaptive predictor

- Many design options!
- One way is to simulate many tables by merging two indexes into one via xor
  - May cause collisions!
- Other methods include concatenating pc and BHT entry, etc
Software for branch predictors

- Software can be optimized with knowledge about branch predictors
  - Usually not by hand! Compiler assist!

- Programmer may expect an if or while statement to usually branch, or usually loop
  - Convey this information to the compiler, so that compiler can optimize code for it
  - e.g., Usually taken branch can be placed on the “branch not taken” path, and more

- GCC has “long __builtin_expect(long exp, long c)”
  - “Expect that exp will usually validate to c”
  - if ( x == 0 ) then foo(); -> if ( __builtin_expect(x==0, 1) ) foo();
    - Expect x==0 to be usually true!

C++20 has [[likely]] [[unlikely]] attributes to tell compiler
Aside:
Intel software for branch predictors

- x86 has a 1-byte opcode prefix to give hints to processor
  - Prefixing a branch instruction with a “0x3E” means it will likely take branch
  - Taken/Not taken

- Unfortunately, deemed not very useful or widely used, and appears to be deprecated in hardware
  - Still valid instruction format, but ignored
  - Wasting one byte without any effect
Aside: Loop unrolling
A compiler solution to branch hazards

```
for ( i = 0 to 15 ) foo();
```

Loop unrolling

```
for ( i = 0 to 3 ) {
    foo();
    foo();
    foo();
    foo();
}
```

Potentially 16 branch mispredicts
Even without mispredicts,
branch instruction consume 16 cycles

Potentially 4 branch mispredicts
Even without mispredicts,
branch instruction consume 4 cycles

Compiler tries its best to be smart about this
- One of key optimizations of gcc’s “-O3”
- Applied automatically if unrolled code is no longer than original
- And many more heuristics/rules!
Some loose topics
(Before getting started on memory systems)

- Microprogramming
  - Now seems to mean a combination of two different things!
Microprogramming of old

Single “Control Logic” responsible for generating “control signals” orchestrating operation at each cycle

An old way of designing/describing microprocessor operation
Microprogramming of old

- Control logic operation described as a Finite State Machine (FSM)
  - Next state depends on current state, and input to the control logic
  - Control signal output depends on current state of the FSM
Microprogramming of old

- Control logic FSM implemented via ROM or PLA
  - “Microprogramming”

Programmable Logic Array (PLA)

RISC processors don’t typically need this
(Simple control logic)
Aside: Microcode and bug patches

- Modern CPUs have programmable portion of the microcode storage
  - No longer entirely ROM
  - Programmable portion takes precedence over original microcode
  - Makes live bug patches possible!
  - Implement same x86 instruction using a different (“bug free”) sequence of microcode operations

- For example, CPU patches for the infamous Spectre exploit involved microcode patches
  - When “BIOS updates” are required, this is often what’s happening
Microprogramming of new: CISC and x86

- x86 ISA is CISC ("Complex")

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>ret</td>
</tr>
<tr>
<td>48 b8 88 77 66 55</td>
<td>movabs rax,0x1122334455667788</td>
</tr>
<tr>
<td>44 33 22 11</td>
<td></td>
</tr>
<tr>
<td>64 ff 03</td>
<td>DWORD PTR fs:[ebx]</td>
</tr>
<tr>
<td>64 67 66 f0 ff 07</td>
<td>lock inc WORD PTR fs:[bx]</td>
</tr>
<tr>
<td>2e c4 e2 71 96 84</td>
<td>vfmaddsub132ps xmm0, xmm1, xmmword ptr cs:</td>
</tr>
<tr>
<td>be 34 23 12 01</td>
<td>[esi + edi * 4 + 0x11223344]</td>
</tr>
</tbody>
</table>

Philipp Koppe et.al., “Reverse Engineering x86 Processor Microcode,” USENIX security 2017
Microprogramming of new: CISC and x86

- Modern microarchitectural advances are difficult to get right on CISC architectures
  - Superscalar, Out-of-Order, Transactional memory, etc
  - Too many conditions and states to keep track of!
- Instead, modern CISC processors internally implement a RISC core with modern bells and whistles
  - e.g., AMD’s patented RISC86 ISA
  - “Front-end” x86 ISA translated by CPU hardware on-the-fly to RISC instructions

```assembly
pop [ebx]  load temp, [esp]
store [ebx], temp
add esp, 4
```

Philipp Koppe et al., “Reverse Engineering x86 Processor Microcode,” USENIX security 2017
Microprogramming complex instructions

- There is typically a fixed sequence of control signals/RISC instructions to generate for one CISC instruction
  - “Microprogram counter” is usually increased by 1
  - Reduces the number of state registers, simplifying microcode memory
- This is not exclusive to CISC-RISC translation. Idea is old!

Microcode decoder is like a small CPU, with PC and everything!
Microprogramming of new

- Microprogramming can be used to generate a sequence of control signals per input instruction
  - Implemented via a chain of FSM states in the control logic
  - No longer designed manually though! Lots of tool research into efficient microcode compilation
  - Usually multiple “decoders” operating in parallel
- We know traditional techniques are still used

Philipp Koppe et.al., “Reverse Engineering x86 Processor Microcode,” USENIX security 2017
Some loose topics
(Before getting started on memory systems)

- Superscalar
  - Just a taste!
Superscalar Processing

- An ideally pipelined processor can handle up to one instruction per cycle
  - Instructions Per Cycle (IPC) = 1, Cycles Per Instruction (CPI) = 1

- Superscalar wants to process multiple instructions per cycle
  - IPC > 1, CPI < 1
  - An N-way superscalar processor handles N instructions per cycle
  - Requires multiple pipeline hardware instances/resources
  - Hardware performs dependency checking on-the-fly between concurrently-fetched instructions
Pipeline for superscalar processing

- Multiple copies of the datapath supports multiple instructions/cycle
- Register file needs many more ports
- Actually requires a complex scheduler in the decode stage!
Superscalar has concurrent hazards

- What if two concurrently issued instructions have dependencies?
  - No choice but to stall the dependent instruction…
  - … in an in-order pipeline! ← Topic for another day

- Data hazards
  - e.g., “addi s1, s0, 1” and “addi s2, s1, 1” issued at the same time?
    - Forwarding won’t work here! Both instructions in decode stage at the same time
    - Scheduler must stagger “addi s2, s1, 1”, sacrificing performance

- Control hazards
  - e.g., How to handle a beq, followed by another instruction?
    - Branch prediction, as usual

Results in very complex control logic! (Chip resources/cost!)
In-order superscalar example

Ideal IPC = 2 (2-Way superscalar)

lw  t0, 40($s0)
add t1, $s1, $s2
sub t2, s1, s3
and t3, s3, s4
or  t4, s1, s5
sw  s5, 80($s0)

No dependencies between any instructions

Actual IPC = 2 (6 instructions issued in 3 cycles)

In-order superscalar with dependencies

Ideal IPC = 2 (2-Way superscalar)

lw  t0, 40(s0)
add t1, t0,$s1
sub t0, s2, s3
and t2, s4, t0
or  t3, s5, s6
sw  s7, 80(t3)

Dependencies across many instructions!

Stalled until writeback can be forwarded
Stalled until exec can be forwarded
No stall, exec can be forwarded

Actual IPC = 1.2 (6 instructions issued in 5 cycles)

In the real-world: Core i7 performance

- Core i7 has a 4-way Out-of-Order Superscalar pipeline
  - Ideally, 0.25 Cycles Per Instruction (CPI)
  - Dependencies and misprediction typically results in much lower performance

Is it worth it? Or do we want just more, simpler cores? Depends on your target area (servers? phones?) and profiling results...
In the real-world: Core i7 performance

- Branch predictors work pretty well!
  - But deep/wide pipelines result in high mispredict overhead