Control Dependencies

- Branches are very frequent
  - Approx. 20% of all instructions
- Can not wait until we know where it goes
  - Long pipelines
    - Branch outcome known after B cycles
    - No scheduling past the branch until outcome known
  - Superscalars (e.g., 4-way)
    - Branch every cycle or so!
    - One cycle of work, then bubbles for ~B cycles?
What to do with branches?

- Eliminate branches
  - Predication (more on later)
- Delayed branch slot
  - SPARC, MIPS
- Or predict? 😊
Surviving Branches: Prediction

• Predict Branches
  – And predict them well!

• Fetch, decode, etc. on the predicted path
  – Option 1: No execute until branch resolved
  – Option 2: Execute anyway (speculation)

• Recover from mispredictions
  – Restart fetch from correct path
Branch Prediction

• Need to know two things
  – Whether the branch is taken or not (direction)
  – The target address if it is taken (target)

• Direct jumps, Function calls: unconditional branches
  – Direction known (always taken), target easy to compute

• Conditional Branches (typically PC-relative)
  – Direction difficult to predict, target easy to compute

• Indirect jumps, function returns
  – Direction known (always taken), target difficult
Branch Prediction: Direction

• Needed for conditional branches
  – Most branches are of this type
• Many, many kinds of predictors for this
  – Static: fixed rule, or compiler annotation
    (e.g. br.bwh (branch whether hint. IA-64))
  – Dynamic: hardware prediction
• Dynamic prediction usually history-based
  – Example: predict direction is the same as the last time this branch was executed
Static Prediction

• Always predict NT
  – easy to implement
  – 30-40% accuracy … not so good

• Always predict T
  – 60-70% accuracy

• BTFNT
  – loops usually have a few iterations, so this is like always predicting that the loop is taken
  – don’t know target until decode
One-Bit Branch Predictor: Last-time predictor

K bits of branch instruction address

Branch history table of $2^K$ entries, 1 bit per entry

Use this entry to predict this branch:

0: predict not taken
1: predict taken

When branch direction resolved, go back into the table and update entry: 0 if not taken, 1 if taken
for(i=0; i < 100000; i++) {
    if( (i % 100) == 0 )
        tick();
    if((i & 1) == 1)
        odd();
}
The Bit Is Not Enough!

• Example: short loop (8 iterations)
  – Taken 7 times, then not taken once
  – Not-taken mispredicted (was taken previously)

Act: TTTTTTTTTTTTTTTTTTTTTNT...
Pred: XTTTTTTNTTTTTTNNTTTTTTTTN
Corr: Xooooo MMooooooMMooooooMM

Misprediction rate: 2/8 = 25%

• Execute the same loop again
  – First always mispredicted (previous outcome was not taken)
  – Then 6 predicted correctly
  – Then last one mispredicted again

• Each fluke/anomaly in a stable pattern results in two mispredicts per loop
Examples

DC08: TTTTTTTTTTT ... TTTTTTTTTTN TTTTTTTTTT ...  
100,000 iterations

How often is branch outcome != previous outcome?
2 / 100,000

DC44: TTTTT ... TTTTTTT ... TTTTTTT ...  
2 / 100

DC50: TNTNTNTNTNTNTNTNTNTNTNTNTNTNT ...  
2 / 2

99.998% Prediction Rate

98.0%

0.0%
Two Bits are Better Than One

- Predict NT
- Predict T
- Transition on T outcome
- Transition on NT outcome

**FSM for Last-time Prediction**

**FSM for 2bC (2-bit Counter)**
Example

1bC:
Initial Training/Warm-up

2bC:

Only 1 Mispredict per N branches now! DC08: 99.999% DC44: 99.0%
Importance of Branches

- 98% ➔ 99%
  - Who cares?
  - Actually, it’s 2% misprediction rate ➔ 1%
  - That’s a halving of the number of mispredictions

- So what?
  - If a pipeline can fetch 5 instructions at a cycle and the branch resolution time is 20 cycles
  - To Fetch 500 instructions
    - 100 accuracy : 100 cycles
    - 98 accuracy:
      - 100 (correctly fetch) + 20 (misprediction)*10 = 300 cycles
    - 99 accuracy
      - 100 (correctly fetch) + 20 misprediction *5 = 200 cycles
Two-level Branch Predictor

BHR
(branch history register)

previous one

index

Pattern History Table

00 .... 00
00 .... 01
00 .... 10
11 .... 11

Yeh&patt'92
Why does Global Predictor Work?

- Branches are correlated

Branch X: if (cond1)

....

Branch Y: if (cond 2)

....

Branch Z: if (cond 1 and cond 2)

<table>
<thead>
<tr>
<th>Branch X</th>
<th>Branch Y</th>
<th>Branch Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Gshare Branch Predictor

Predictor size: $2^{\text{history length}} \times 2\text{bit}$

McFarling’93
Biomodal Branch Predictor

2^n entry table

PC 0x809000

n-bit

Typical Local predictor
When does it work?
- Loop,
- Repeat pattern

```c
a++;
if (!(a%3)) { .. }
```
Target Address Prediction

- Branch Target Buffer
  - IF stage: need to know fetch addr every cycle
  - Need target address one cycle after fetching a branch
  - For some branches (e.g., indirect) target known only after EX stage, which is way too late
  - Even easily-computed branch targets need to wait until instruction decoded and direction predicted in ID stage (still at least one cycle too late)
  - So, we have a fast predictor for the target that only needs the address of the branch instruction
Branch Target Buffer

- BTB indexed by instruction address (or fetch address)
- We don’t even know if it is a branch!
- If address matches a BTB entry, it is predicted to be a branch
- BTB entry tells whether it is taken (direction) and where it goes if taken
- BTB takes only the instruction address, so while we fetch one instruction in the IF stage we are predicting where to fetch the next one from

Direction prediction can be factored out into separate table
Branch Target Buffer

- PC of instruction to fetch
- Look up
- Predicted PC
- Number of entries in branch-target buffer
- No: instruction is not predicted to be branch; proceed normally
- Yes: then instruction is branch and predicted PC should be used as the next PC
- Branch predicted taken or untaken

© 2003 Elsevier Science (USA). All rights reserved.
Function Calls

main()
{
    foo();
    printf("still hungry\n");
    ...
    foo();
    printf("full\n");
}

foo()
{
    ....
    return
}

BTB
Return Address Stack (RAS)

- Function returns are frequent, yet
  - Address is difficult to compute
    (have to wait until EX stage done to know it)
  - Address difficult to predict with BTB
    (function can be called from multiple places)
main() {
    foo();
    printf("still hungry\n");
    foo();
    printf("full\n");
}

foo() {
    ..... return
    }

Function Calls
Return Address Stack (RAS)

- But return address is actually easy to predict
  - It is the address after the last call instruction that we haven’t returned from yet
  - Hence the Return Address Stack
Function Calls

main()
{
  foo();
  printf("still hungry\n");
  ..... foo(); printf("full\n");
}

foo()
{
  return
}

0x904
Return Address Stack (RAS)

- Call pushes return address into the RAS
- When a return instruction decoded, pop the predicted return address from RAS
- Accurate prediction even w/ small RAS
LOOP Branches

```c
for (ii = 0; ii < 10; ii++) {
    ...
}
```

Loop branch is iterated 10 times all the time

- Special treatment for loop branches
Options

• Prepare to branch (HPL-PD)
  – Software gives hints to the hardware about what the branch target will be. It saves us the target prediction since it has already been written into one of the target registers.

• Special Loop predictor (Intel’s Pentium M)
  – Detect a loop branch
  – Train the max iteration counter value
Direct vs. Indirect Branch

Conditional (Direct) Branch
- T
- N
- br.cond TARGET
- TARG
- A+1

Indirect Branch
- A
- α
- β
- δ
- ρ
- R1 = MEM[R2]
- branch R1

• Use the BTB
• A special indirect branch predictor (Intel’s Core-2)
Eliminating Branches

- Predication
- Loop unrolling
Predication

if (cond) {
    b = 0;
} else {
    b = 1;
}

Convert control flow dependency to data dependency

Pro: Eliminate hard-to-predict branches (in traditional architecture)
    Eliminate branch divergence (in CUDA)
Cons: Extra instructions
Instruction Predication in G80

- Comparison instructions set condition codes (CC)
- Instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)
- Compiler tries to predict if a branch condition is likely to produce many divergent warps
  - If guaranteed not to diverge: only predicates if < 4 instructions
  - If not guaranteed: only predicates if < 7 instructions
- May replace branches with instruction predication
- ALL predicated instructions take execution cycles
  - Those with false conditions don’t write their output
  - Or invoke memory loads and stores
  - Saves branch instructions, so can be cheaper than serializing divergent paths
Loop Unrolling

• Transforms an M-iteration loop into a loop with M/N iterations
  – We say that the loop has been unrolled N times

```c
for(i=0; i<100; i+=4) {
    a[i] *= 2;
    a[i+1] *= 2;
    a[i+2] *= 2;
    a[i+3] *= 2;
}
```

Some compilers can do this (gcc -funroll-loops)
Or you can do it manually (above)

http://www.cc.gatech.edu/~milos/CS6290F07/
Why Loop Unrolling? (1)

- Less loop overhead

```c
for(i=0;i<100;i++) a[i] += 2;
```

```c
for(i=0;i<100;i+=4)
    a[i] += 2;
    a[i+1] += 2;
    a[i+2] += 2;
    a[i+3] += 2;
```

How many branches?

http://www.cc.gatech.edu/~milos/CS6290F07/
Why Loop Unrolling? (2)

- Allows better scheduling of instructions

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

http://www.cc.gatech.edu/~milos/CS6290F07/
Why Loop Unrolling? (3)

• Get rid of small loops

```c
for(i=0; i<4; i++) a[i] *= 2;  // a[0] *= 2;  
a[1] *= 2;  // a[1] *= 2;  
```

Difficult to schedule/hoist insts from bottom block to top block due to branches

Easier: no branches in the way

http://www.cc.gatech.edu/~milos/CS6290F07/
• **VLIW** = Very Long Instruction Word


• **Everything** is statically scheduled
  - All hardware resources exposed to compiler
  - Compiler must figure out what to do and when to do it
  - Get rid of complex scheduling hardware
  - More room for “useful” resources

• Examples:
  - Texas Instruments DSP processors
  - Transmeta’s processors
  - Intel IA-64 (EPIC), ATI Graphics processor
Static Instruction Scheduling
Why VLIW is good?

• Let the compiler do all of the hard work
  – Expose functional units, bypasses, latencies, etc.
  – Compiler can do its best to schedule code well
  – Compiler has plenty of time to do analysis
  – Compiler has larger scope (view of the program)

• Works extremely well on regular codes
  – Media Processing, Scientific, DSP, etc.

• Can be energy-efficient
  – Dynamic scheduling hardware is power-hungry
Why is VLIW hard?

- Latencies are not constant
  - Statically scheduled assuming fixed latencies
- Irregular applications
  - Dynamic data structures (pointers)
  - “Common Case” changes when input changes
- Code can be very large
  - *Every resource exposed* also means that instructions are “verbose”, with fields to tell each HW resource what to do
  - Many, many “NOP” fields
- 3wide VLIW machine \(\rightarrow\) 6 wide VLIW machine?
- Where are instruction parallelism?
Extreme Example: Intel IA-64 (EPIC)

• Goal: Keep the best of VLIW, fix problems
  – Keep HW simple and let the compiler do its job
  – Support to deal with non-constant latencies
  – Make instructions more compact

• The reality
  – Compiler still very good at regular codes
  – HW among the most complex ever built by Intel
  – Good news: compiler still improving
SPE in Cell

- VLIWish