Computer Architecture: Branch Prediction (II) and Predicated Execution

Prof. Onur Mutlu
Carnegie Mellon University

A Note on This Lecture

- These slides are partly from 18-447 Spring 2013, Computer Architecture, Lecture 12: Predicated Execution

- Video of that lecture:
    - http://www.youtube.com/watch?v=xtA1arYjq-M

Last Lecture

- Branch prediction

Today’s Agenda

- Wrap up control dependence handling

- State recovery mechanisms, interrupts, exceptions

Control Dependence Handling

Review: How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Review: Importance of The Branch Problem

- Assume a 5-wide *superscalar* pipeline with 20-cycle branch resolution latency

- How long does it take to fetch 500 instructions?
  - Assume no fetch breaks and 1 out of 5 instructions is a branch
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy
    - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
    - 40% extra instructions fetched
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched

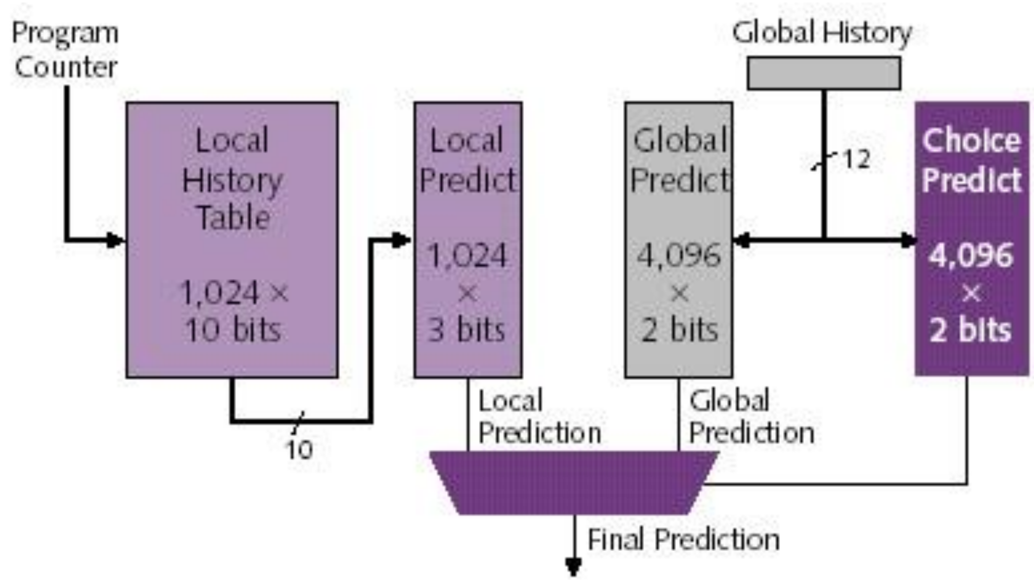
Review: Local and Global Branch Prediction

- Last-time and 2BC predictors exploit "last-time" predictability

- Realization 1: A branch's outcome can be correlated with other branches' outcomes
  - Global branch correlation

- Realization 2: A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)
  - Local branch correlation

Review: Hybrid Branch Prediction in Alpha 21264



- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Review: Predicate Combining (*not* Predicated Execution)

- Complex predicates are converted into multiple branches
  - if ((a == b) && (c < d) && (a > 5000)) { … }
    - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate

\+ Fewer branches in code → fewer mipredictions/stalls

-- Possibly unnecessary work

-- If the first predicate is false, no need to compute other predicates

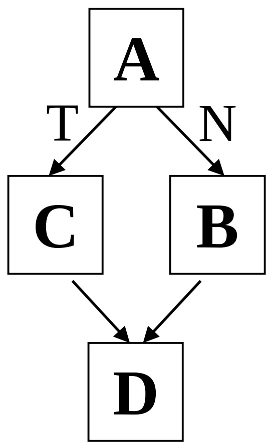
- Condition registers exist in IBM RS6000 and the POWER architecture

Predication (Predicated Execution)

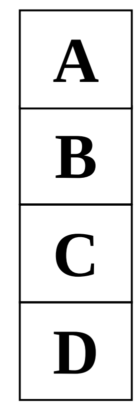
- Idea: Compiler converts control dependence into data dependence → branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (others turned into NOPs)

(normal branch code) (predicated code)

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




```
A   p1 = (cond)
    branch p1, TARGET
B   mov b, 1
    jmp JOIN
C   TARGET:
    mov b, 0
D   add  x, b, 1
```




```
A   p1 = (cond)
B   (!p1) mov  b, 1
C   (p1) mov  b, 0
D   add  x, b, 1
```

Conditional Move Operations

- Very limited form of predicated execution

- CMOV R1 $\leftarrow$ R2
  - R1 = (ConditionCode == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

Review: CMOV Operation

- Suppose we had a Conditional Move instruction…
  - CMOV condition, R1 ← R2
  - R1 = (condition == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

- Code example with branches vs. CMOVs

  if (a == 5) {b = 4;} else {b = 3;}

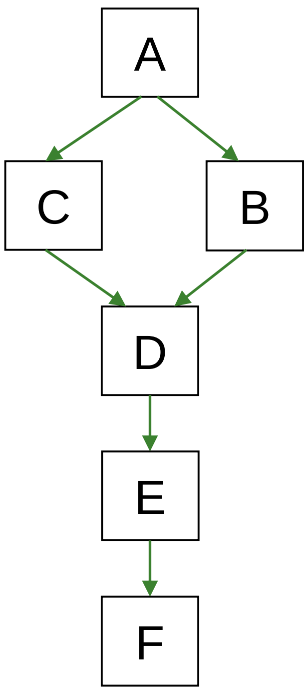
  CMPEQ condition, a, 5;

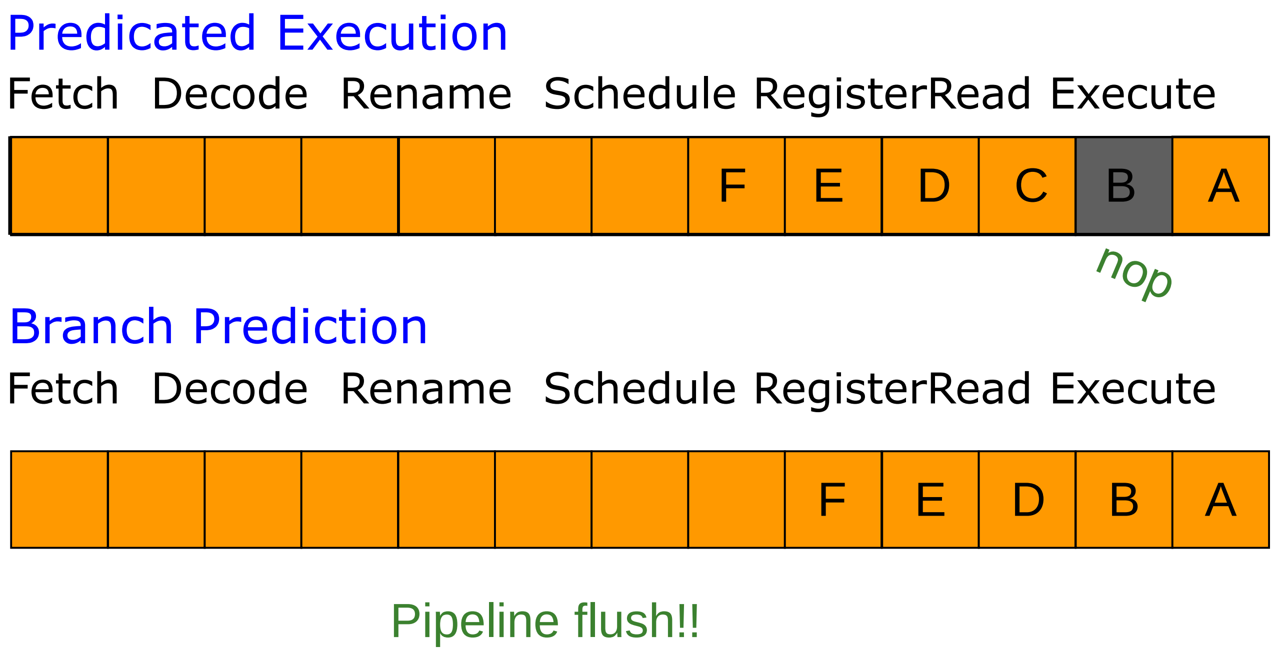
  CMOV condition, b ← 4;

  CMOV !condition, b ← 3;

Predicated Execution (II)

- Predicated execution can be high performance and energy-efficient





Predicated Execution (III)

- Advantages:

  + Eliminates mispredictions for hard-to-predict branches

    + No need for branch prediction for some branches

    + Good if misprediction cost > useless work due to predication

  + Enables code optimizations hindered by the control dependency

    + Can move instructions more freely within predicated code

- Disadvantages:

  -- Causes useless work for branches that are easy to predict

    -- Reduces performance if misprediction cost < useless work

    -- Adaptivity: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, phase, control-flow path.

  -- Additional hardware and ISA support

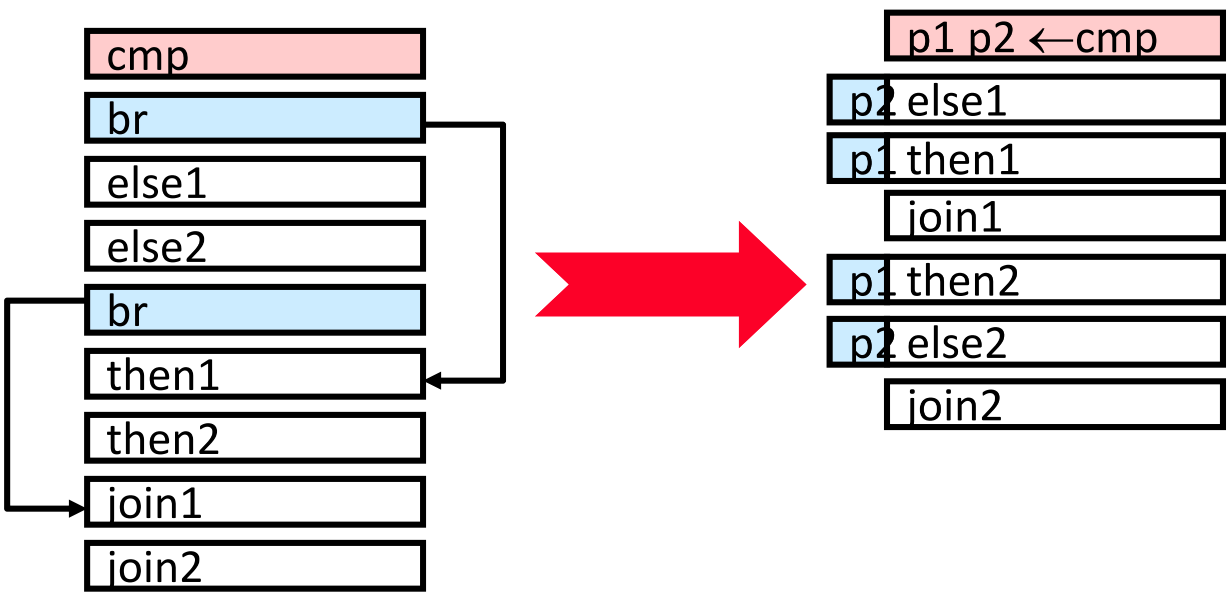
  -- Cannot eliminate all hard to predict branches

    -- Loop branches?

Predicated Execution in Intel Itanium

- Each instruction can be separately predicated
- 64 one-bit predicate registers

  each instruction carries a 6-bit predicate field
- An instruction is effectively a NOP if its predicate is false



Conditional Execution in ARM ISA

- Almost all ARM instructions can include an optional condition code.

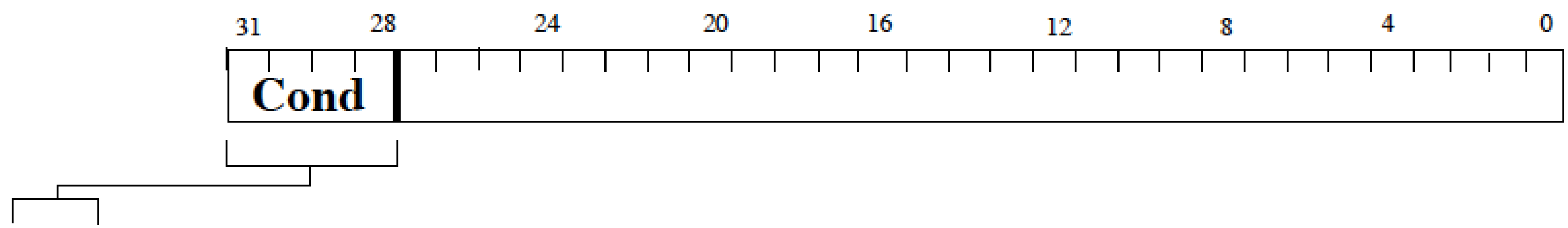
- An instruction with a condition code is only executed if the condition code flags in the CPSR meet the specified condition.

Conditional Execution in ARM ISA

31 28 27 16 15 8 7 0

															Instruction type
Cond	0 0	I	Opcode	S	Rn	Rd	Operand2								Data processing / PSR Transfer
Cond	0 0 0 0 0 0	A	S	Rd	Rn	Rs	1 0 0 1	Rm							Multiply
Cond	0 0 0 0 1	U	A	S	RdHi	RdLo	Rs	1 0 0 1	Rm						Long Multiply **(v3M / v4 only)**
Cond	0 0 0 1 0	B	0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm							Swap
Cond	0 1	I	P	U	B	W	L	Rn	Rd	Offset					Load/Store Byte/Word
Cond	1 0 0	P	U	S	W	L	Rn	Register List							Load/Store Multiple
Cond	0 0 0	P	U	1	W	L	Rn	Rd	Offset1	1	S	H	1	Offset2	Halfword transfer : Immediate offset (v4 only)
Cond	0 0 0	P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm	Halfword transfer: Register offset (v4 only)
Cond	1 0 1	L	Offset												Branch
Cond	0 0 0 1	0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn								Branch Exchange **(v4T only)**
Cond	1 1 0	P	U	N	W	L	Rn	CRd	CPNum	Offset					Coprocessor data transfer
Cond	1 1 1 0	Op1	CRn	CRd	CPNum	Op2	0	CRm							Coprocessor data operation
Cond	1 1 1 0	Op1	L	CRn	Rd	CPNum	Op2	1	CRm						Coprocessor register transfer
Cond	1 1 1 1	SWI Number													Software interrupt

Conditional Execution in ARM ISA



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

Conditional Execution in ARM ISA

* **To execute an instruction conditionally, simply postfix it with the appropriate condition:**
  - For example an add instruction takes the form:

```
– ADD r0,r1,r2          ; r0 = r1 + r2 (ADDAL)
```

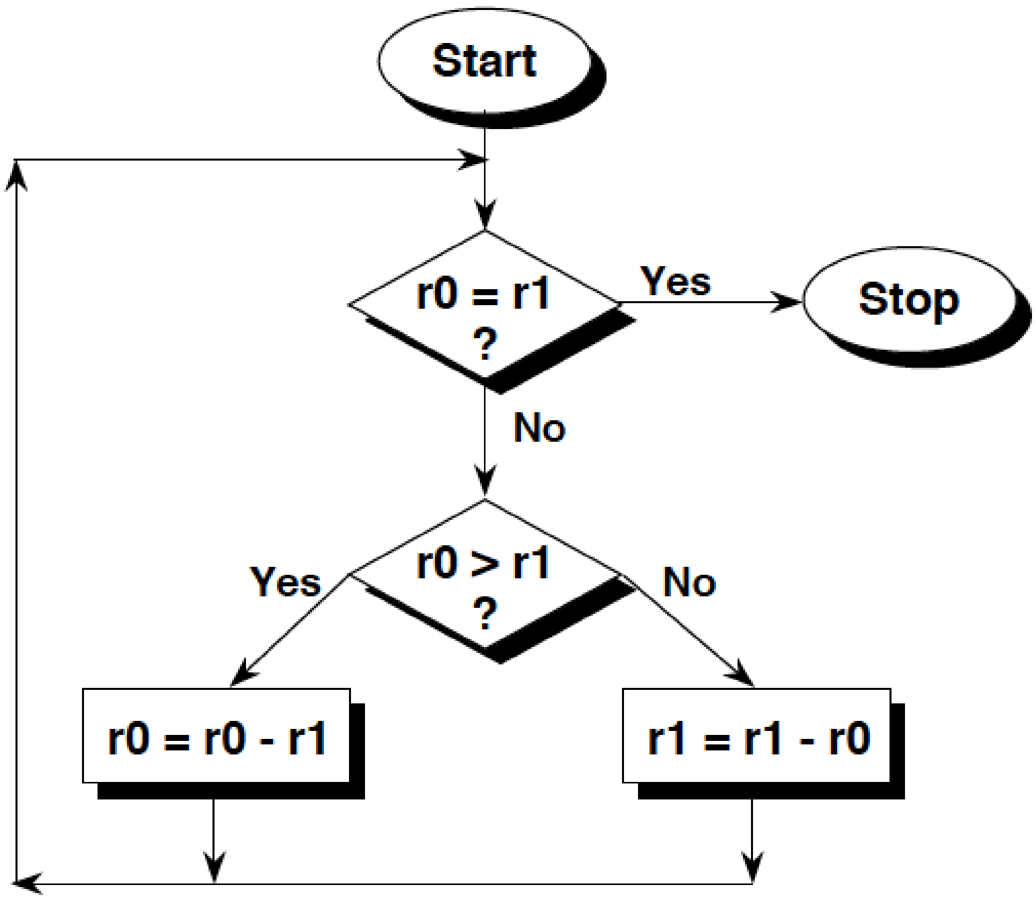
  - To execute this only if the zero flag is set:

```
– ADDEQ r0,r1,r2        ; If zero flag set then…
                        ; ... r0 = r1 + r2
```

* **By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an "S".**
  - For example to add two numbers and set the condition flags:

```
– ADDS r0,r1,r2         ; r0 = r1 + r2
                        ; ... and set flags
```

Conditional Execution in ARM ISA



* **Convert the GCD algorithm given in this flowchart into**
  1) "Normal" assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.

* **The only instructions you need are CMP, B and SUB.**

Conditional Execution in ARM ISA

"Normal" Assembler

```
gcd     cmp r0, r1       ;reached the end?
        beq stop
        blt less         ;if r0 > r1
        sub r0, r0, r1   ;subtract r1 from r0
        bal gcd
less    sub r1, r1, r0   ;subtract r0 from r1
        bal gcd
stop
```

ARM Conditional Assembler

```
gcd     cmp   r0, r1       ;if r0 > r1
        subgt r0, r0, r1   ;subtract r1 from r0
        sublt r1, r1, r0   ;else subtract r0 from r1
        bne   gcd          ;reached the end?
```

Idealism

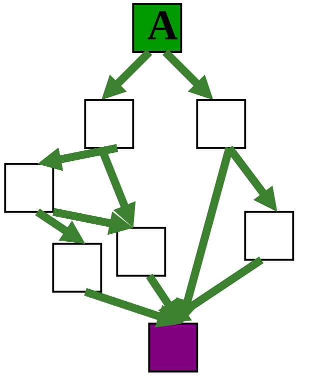
- Wouldn't it be nice
  - If the branch is eliminated (predicated) when it will actually be mispredicted
  - If the branch were predicted when it will actually be correctly predicted

- Wouldn't it be nice
  - If predication did not require ISA support

Improving Predicated Execution

- Three major limitations of predication
  1. Adaptivity: non-adaptive to branch behavior
  2. Complex CFG: inapplicable to loops/complex control flow graphs
  3. ISA: Requires large ISA changes

- Wish Branches [Kim+, MICRO 2005]
  - Solve 1 and partially 2 (for loops)



- Dynamic Predicated Execution
  - Diverge-Merge Processor [Kim+, MICRO 2006]
    - Solves 1, 2 (partially), 3

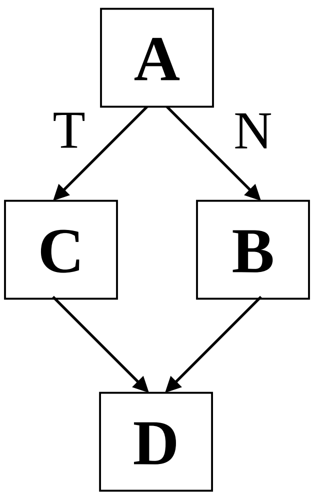
Wish Branches

- The **compiler** generates code (with wish branches) that can be executed **either** as predicated code **or** non-predicated code (normal branch code)
- The **hardware decides** to execute predicated code or normal branch code at run-time based on the confidence of branch prediction
- **Easy to predict: normal branch code**
- **Hard to predict: predicated code**

- Kim et al., "Wish Branches: Enabling Adaptive and Aggressive Predicated Execution," MICRO 2006, IEEE Micro Top Picks, Jan/Feb 2006.

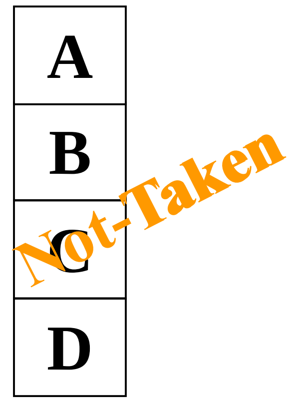
Wish Jump/Join

High Confidence



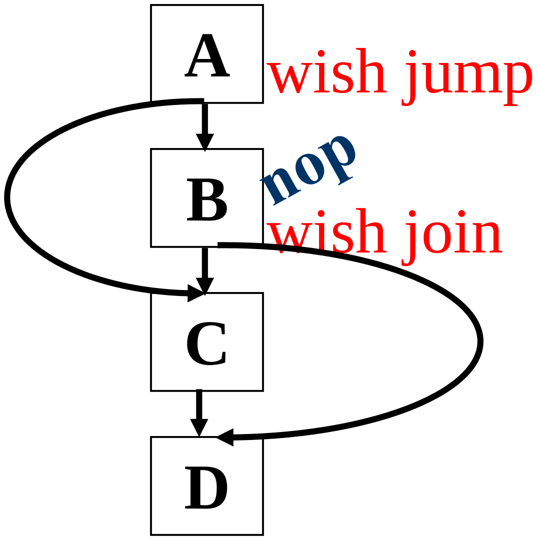
A	
	p1 = (cond) branch p1, TARGET
B	mov b, 1 jmp JOIN
C	TARGET: mov b,0

normal branch code



A	p1 = (cond)
B	(!p1) mov b,1
C	(p1) mov b,0

predicated code



A	p1=(cond) wish.jump p1 TARGET
B	(!p1) mov b,1 wish.join !p1 JOIN
C	TARGET: (p1) mov b,0
D	**JOIN:**

nop

wish jump/join code

Wish Branches vs. Predicated Execution

- Advantages compared to predicated execution
  - **Reduces the overhead** of predication
  - Increases the benefits of predicated code by allowing the compiler to generate more **aggressively-predicated code**
  - Makes predicated code less dependent on machine configuration (e.g. branch predictor)

- Disadvantages compared to predicated execution
  - Extra branch instructions use machine resources
  - Extra branch instructions increase the contention for branch predictor table entries
  - Constrains the compiler's scope for **code optimizations**

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Multi-Path Execution

- Idea: Execute both paths after a conditional branch
  - For all branches: Riseman and Foster, "The inhibition of potential parallelism by conditional jumps," IEEE Transactions on Computers, 1972.
  - For a hard-to-predict branch: Use dynamic confidence estimation

- Advantages:

  + Improves performance if misprediction cost > useless work

  + No ISA change needed

- Disadvantages:

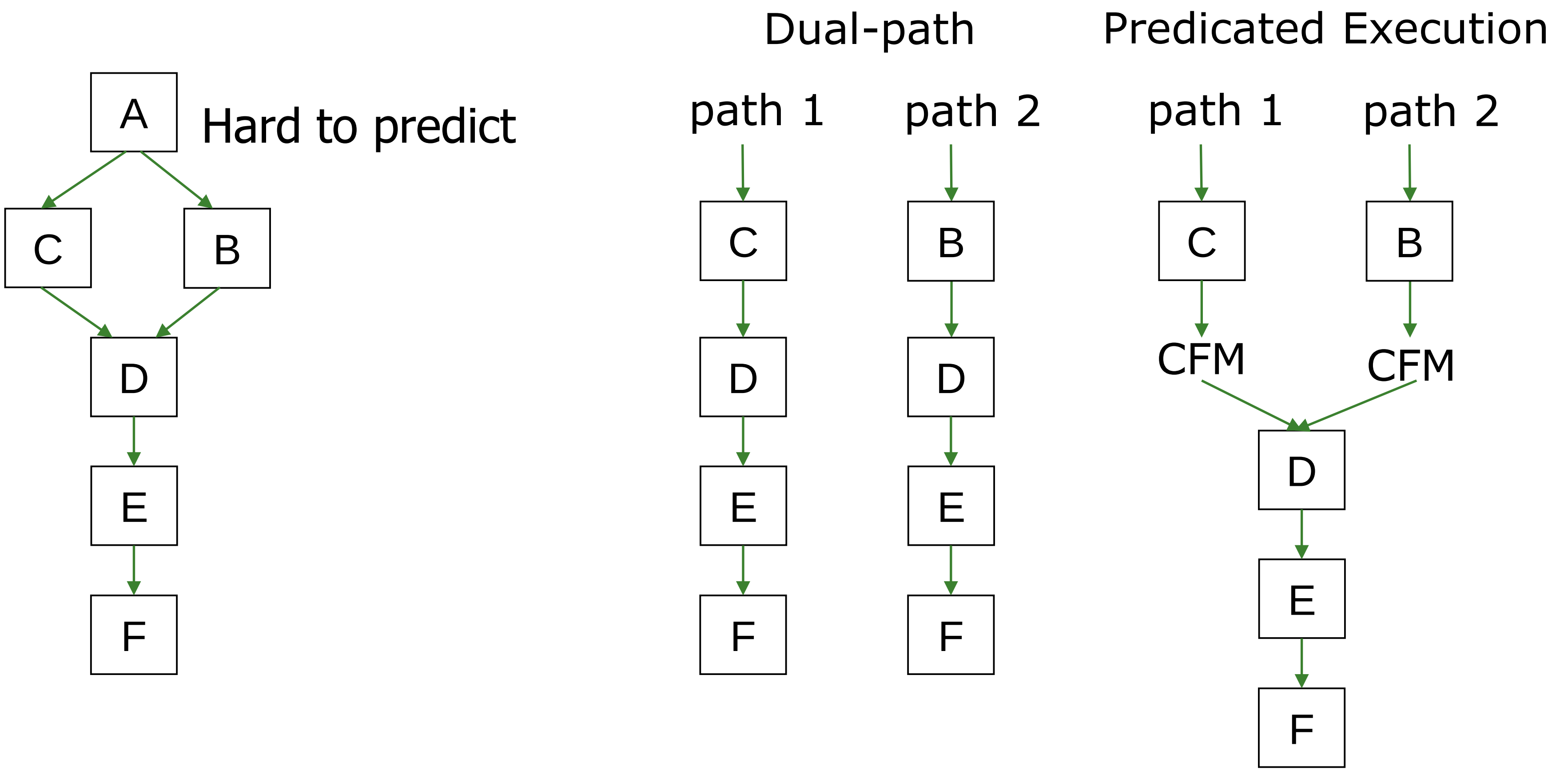
  -- What happens when the machine encounters another hard-to-predict branch? Execute both paths again?

    -- Paths followed quickly become exponential

  -- Each followed path requires its own registers, PC, GHR

  -- Wasted work (and reduced performance) if paths merge

Dual-Path Execution versus Predication



Remember: Branch Types

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

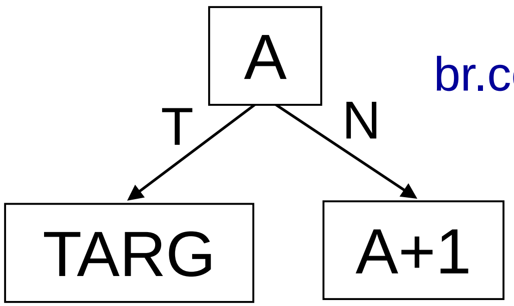
Call and Return Prediction

- Direct calls are easy to predict
  - Always taken, single target
  - Call marked in BTB, target predicted by BTB
- Returns are indirect branches
  - A function can be called from many points in code
  - A return instruction can have many target addresses
    - Next instruction after each call point for the same function
  - Observation: Usually a return matches a call
  - Idea: Use a stack to predict return addresses (Return Address Stack)
    - A fetched call: pushes the return (next instruction) address on the stack
    - A fetched return: pops the stack and uses the address as its predicted target
    - Accurate most of the time: 8-entry stack → > 95% accuracy

```
Call X
    …
    Call X
        …
        Call X
        …
        Return
    Return
Return
```

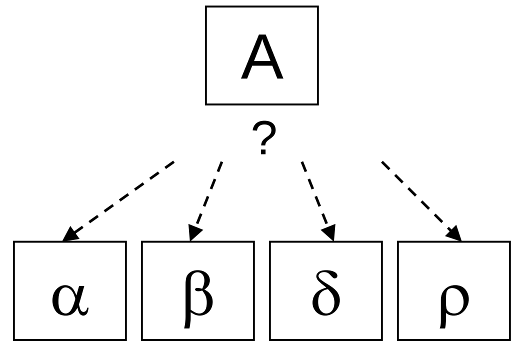
Indirect Branch Prediction (I)

- Register-indirect branches have multiple targets



br.cond TARGET

Conditional (Direct) Branch



R1 = MEM[R2]
branch R1

Indirect Jump

- Used to implement
  - Switch-case statements
  - Virtual function calls
  - Jump tables (of function pointers)
  - Interface calls

Indirect Branch Prediction (II)

- No direction prediction needed
- Idea 1: Predict the last resolved target as the next fetch address

  + Simple: Use the BTB to store the target address

  -- Inaccurate: 50% accuracy (empirical). Many indirect branches switch between different targets

- Idea 2: Use history based target prediction
  - E.g., Index the BTB with GHR XORed with Indirect Branch PC
  - Chang et al., "Target Prediction for Indirect Jumps," ISCA 1997.

  + More accurate

  -- An indirect branch maps to (too) many entries in BTB

  -- Conflict misses with other branches (direct or indirect)

  -- Inefficient use of space if branch has few target addresses

Issues in Branch Prediction (I)

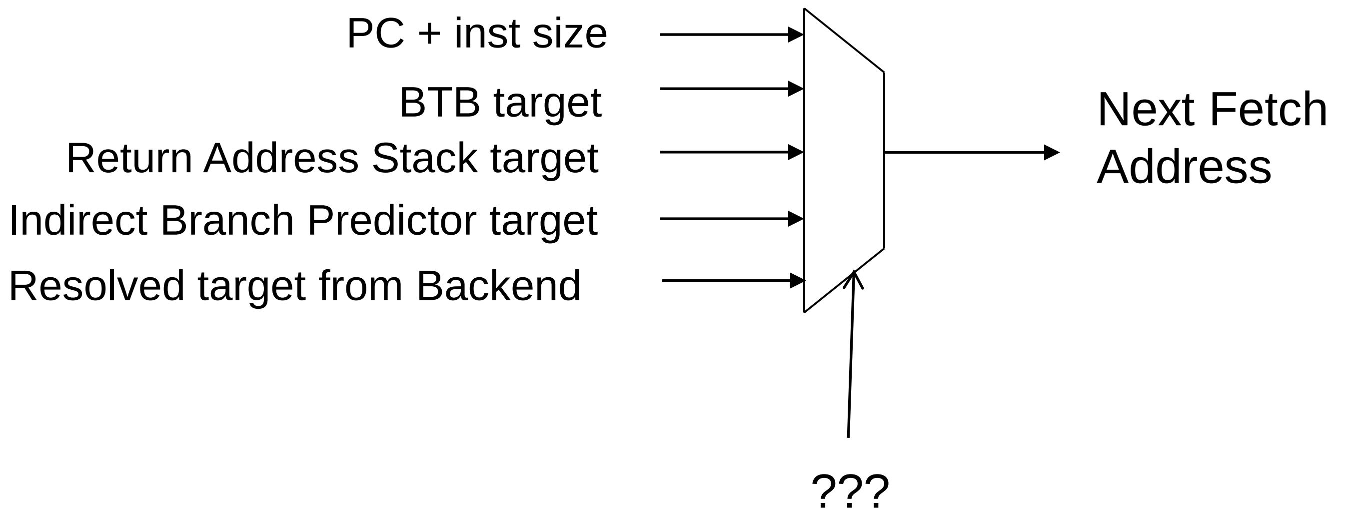
- Need to identify a branch before it is fetched

- How do we do this?
  - BTB hit → indicates that the fetched instruction is a branch
  - BTB entry contains the "type" of the branch

- What if no BTB?
  - Bubble in the pipeline until target address is computed
  - E.g., IBM POWER4

Issues in Branch Prediction (II)

- Latency: Prediction is latency critical
  - Need to generate next fetch address for the next cycle
  - Bigger, more complex predictors are more accurate but slower



Complications in Superscalar Processors

- "Superscalar" processors
  - attempt to execute more than 1 instruction-per-cycle
  - must fetch multiple instructions per cycle

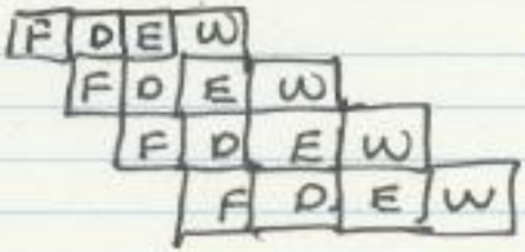
- Consider a 2-way superscalar fetch scenario

  (case 1) Both insts are not taken control flow inst
    - nPC = PC + 8

  (case 2) One of the insts is a <u>taken</u> control flow inst
    - nPC = predicted target addr
    - *NOTE* both instructions could be control-flow; prediction based on the first one predicted taken
    - If the 1st instruction is the predicted taken branch

      → nullify 2nd instruction fetched

Multiple Instruction Fetch: Concepts



Fetch 1 inst/cycle

- Downside:
  Flynn's bottleneck
  If you fetch 1 inst/cycle you cannot finish >1 inst /cycle

Fetch 4 inst/cycle

Two major approaches

1) VLIW

Compiler decides what insts. can be executed in parallel
→ Simple hardware

2) Superscalar

Hardware detects dependencies between instructions that are fetched in the same cycle.

Review of Last Few Lectures

- Control dependence handling in pipelined machines
  - Delayed branching
  - Fine-grained multithreading
  - Branch prediction
    - Compile time (static)
      - Always NT, Always T, Backward T Forward NT, Profile based
    - Run time (dynamic)
      - Last time predictor
      - Hysteresis: 2BC predictor
      - Global branch correlation → Two-level global predictor
      - Local branch correlation → Two-level local predictor
  - Predicated execution
  - Multipath execution