

Systems I

Pipelining V

Topics

- Branch prediction
- State machine design

Branch Prediction

Until now - we have assumed a “predict taken” strategy for conditional branches

- Compute new branch target and begin fetching from there
- If prediction is incorrect, flush pipeline and begin refetching

However, there are other strategies

- Predict not-taken
- Combination (quasi-static)
 - Predict taken if branch backward (like a loop)
 - Predict not taken if branch forward

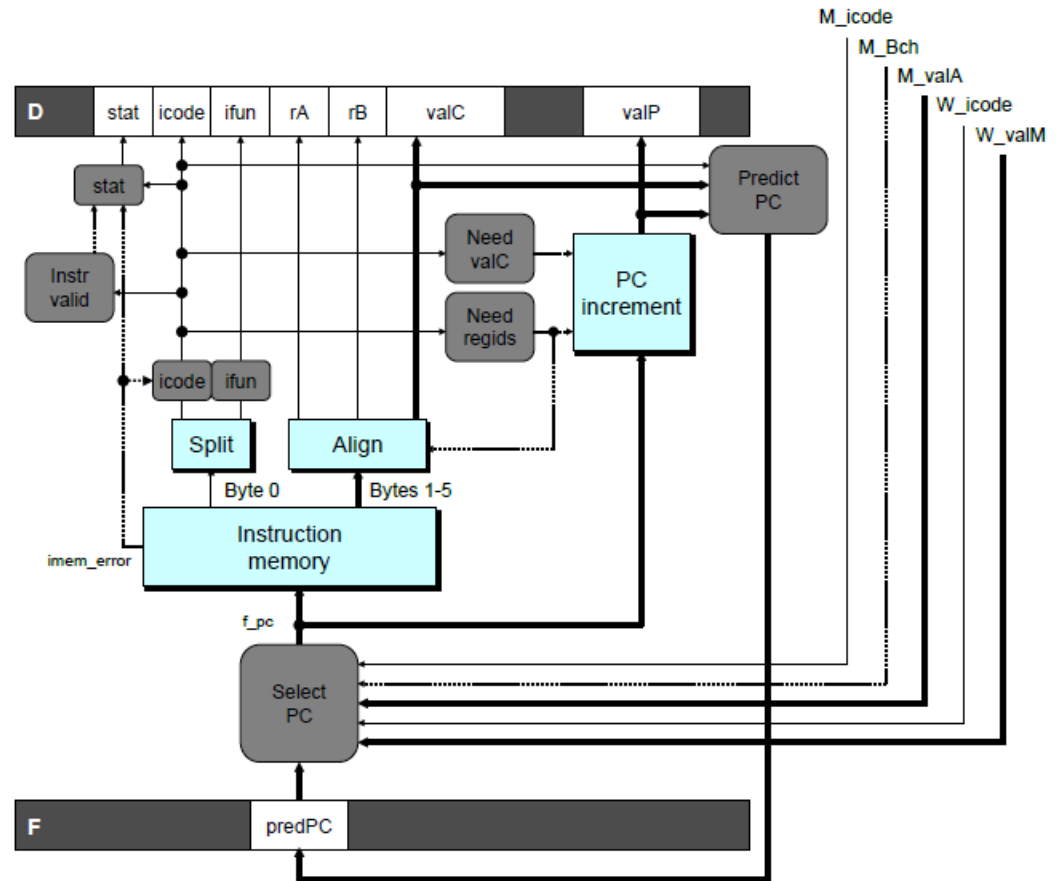
Fetch Logic Revisited

During Fetch Cycle

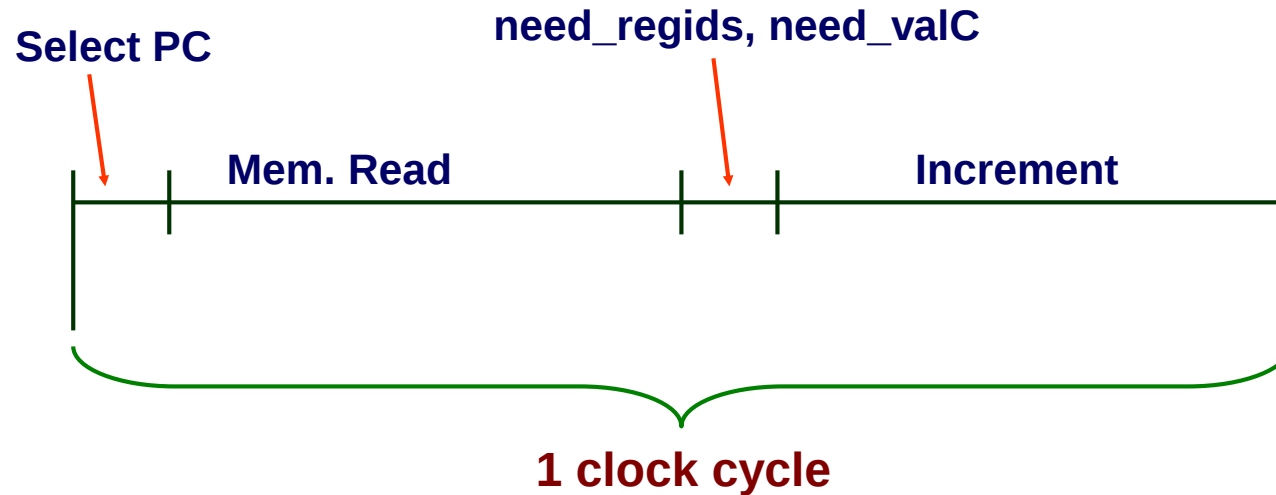
1. Select PC
2. Read bytes from instruction memory
3. Examine icode to determine instruction length
4. Increment PC

Timing

- Steps 2 & 4 require significant amount of time

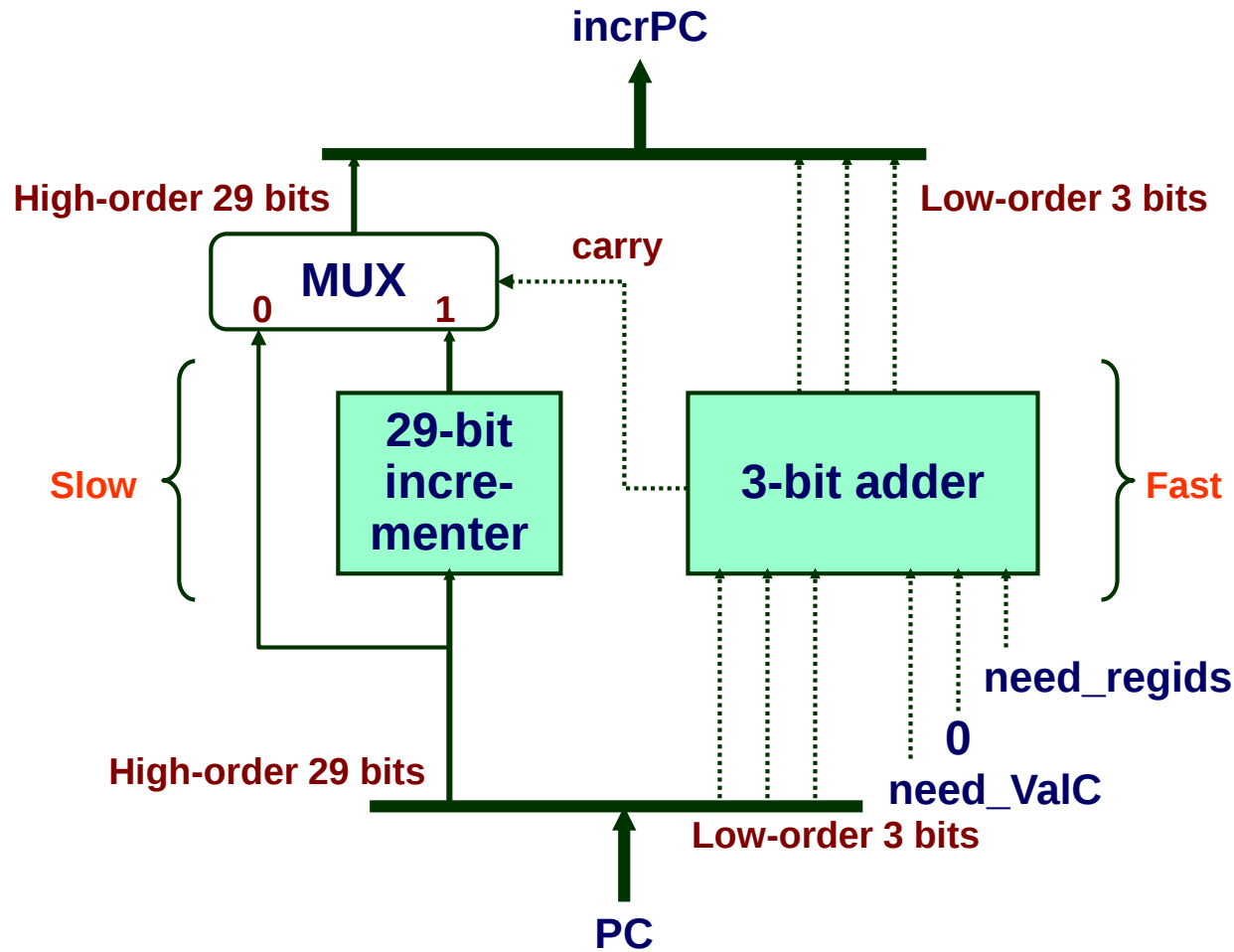


Standard Fetch Timing

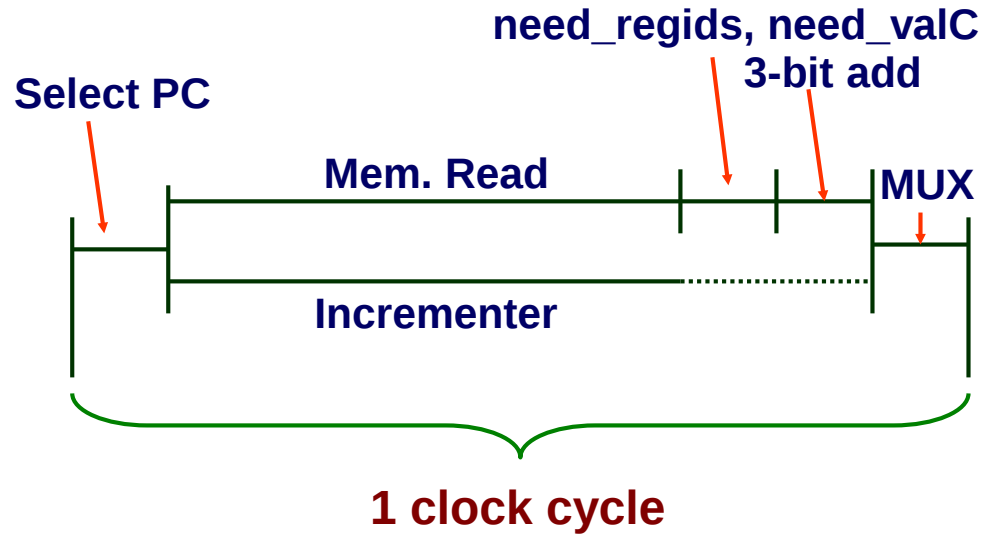


- Must Perform Everything in Sequence
- Can't compute incremented PC until know how much to increment it by

A Fast PC Increment Circuit



Modified Fetch Timing

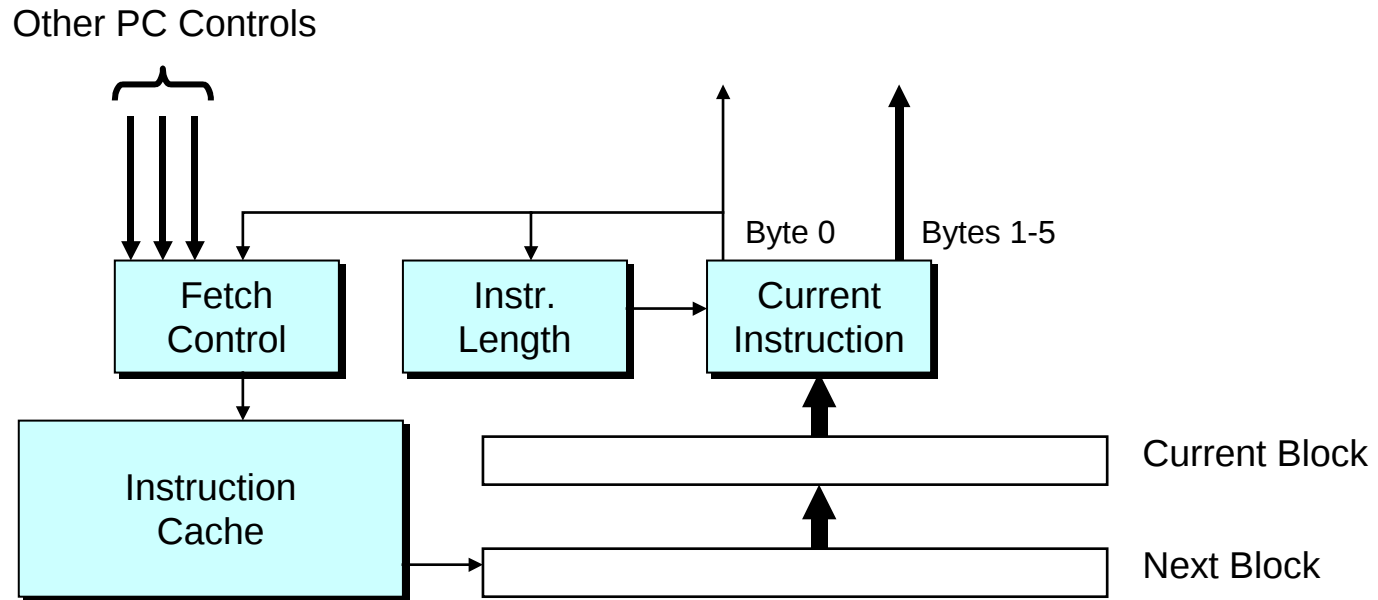


Standard cycle

29-Bit Incrementer

- Acts as soon as PC selected
- Output not needed until final MUX
- Works in parallel with memory read

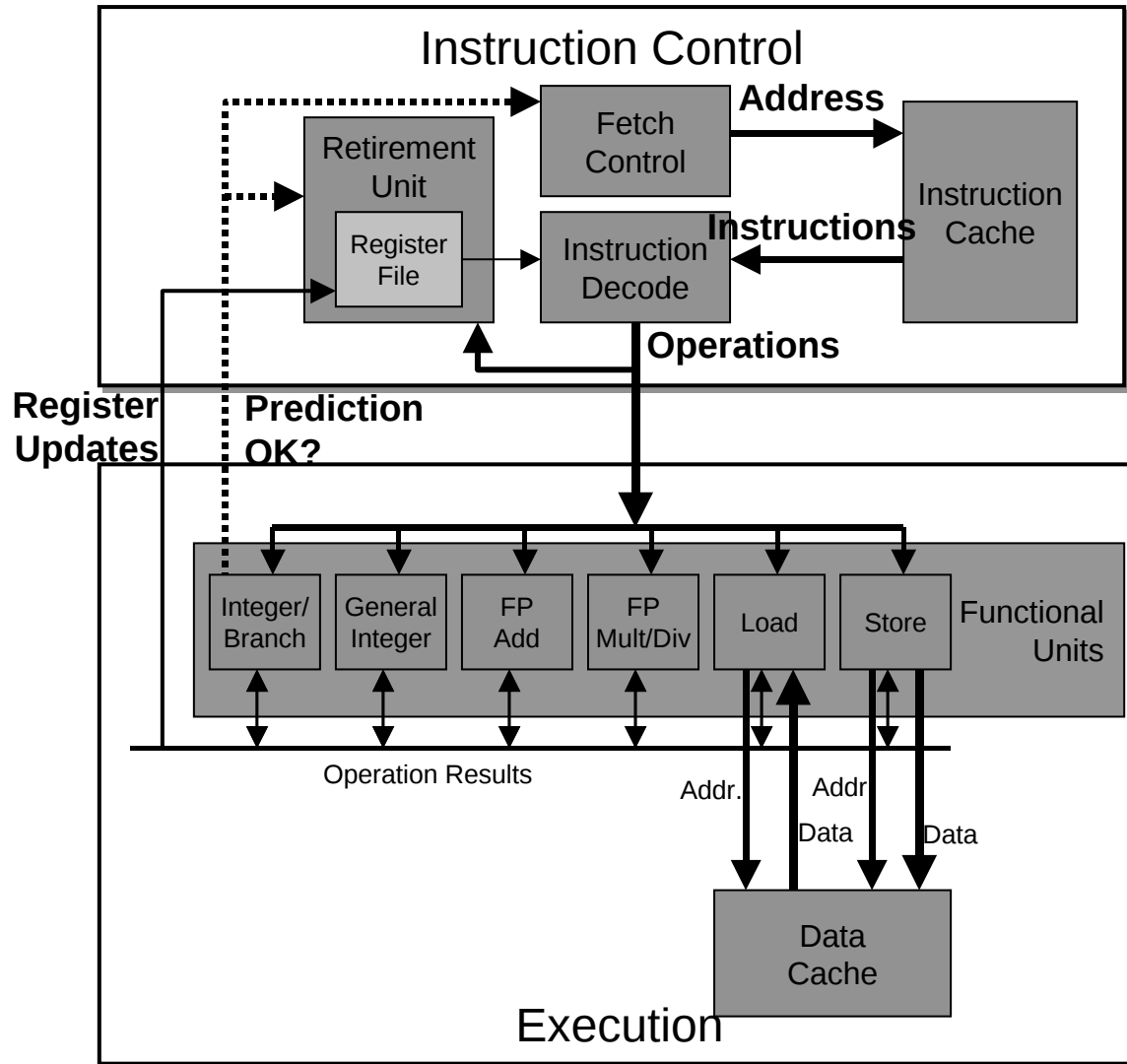
More Realistic Fetch Logic



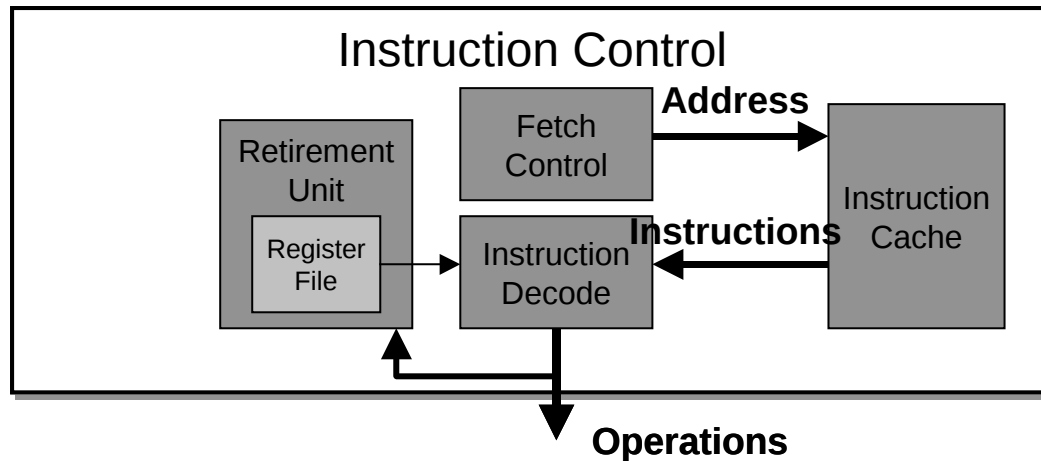
Fetch Box

- Integrated into instruction cache
- Fetches entire cache block (16 or 32 bytes)
- Selects current instruction from current block
- Works ahead to fetch next block
 - As reaches end of current block
 - At branch target

Modern CPU Design



Instruction Control



Grabs Instruction Bytes From Memory

- Based on Current PC + Predicted Targets for Predicted Branches
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

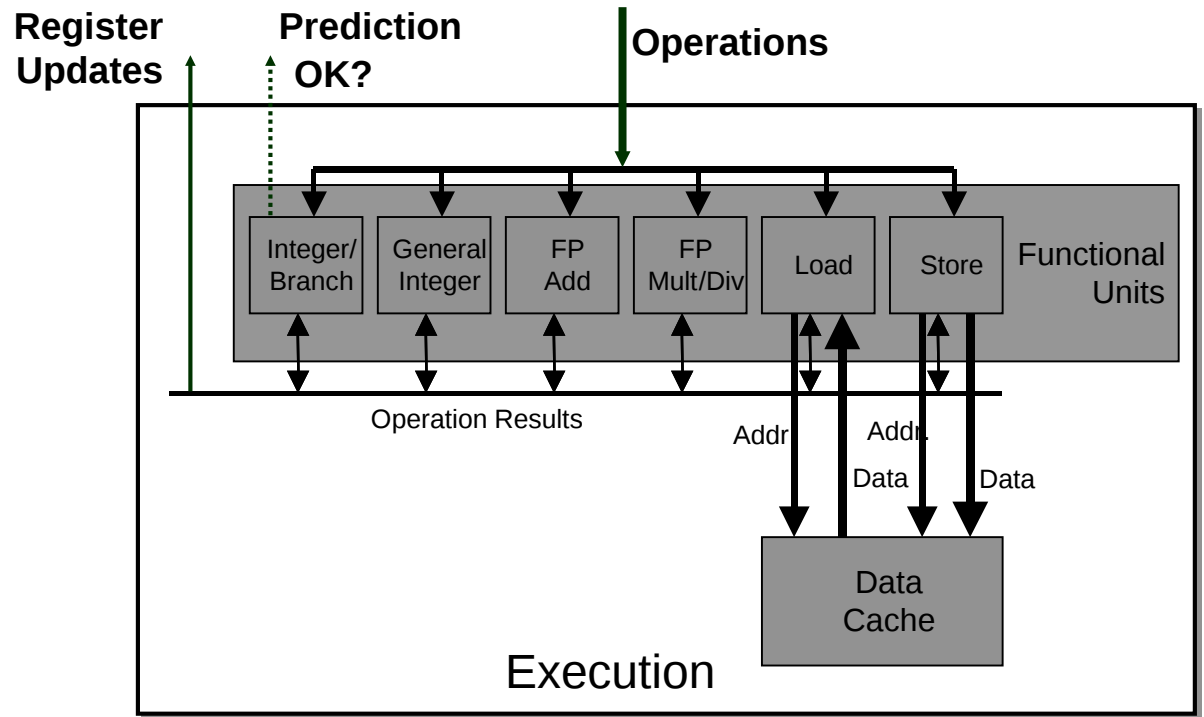
Translates Instructions Into *Operations*

- Primitive steps required to perform instruction
- Typical instruction requires 1–3 operations

Converts Register References Into *Tags*

- Abstract identifier linking destination of one operation with sources of later operations

Execution Unit



- **Multiple functional units**
 - Each can operate independently
- **Operations performed as soon as operands available**
 - Not necessarily in program order
 - Within limits of functional units
- **Control logic**
 - Ensures behavior equivalent to sequential program execution

CPU Capabilities of Intel iCore7

Multiple Instructions Can Execute in Parallel

- 1 load
- 1 store
- 1 FP multiplication or division
- 1 FP addition
- > 1 integer operation

Some Instructions Take > 1 Cycle, but Can be Pipelined

■ Instruction	Latency	Cycles/Issue
■ Load / Store	3	1
■ Integer Multiply	3	1
■ Integer Divide	11—21	5—13
■ Double/Single FP Multiply	4	1
■ Double/Single FP Add	3	1
■ Double/Single FP Divide	10—15	6—11

iCore Operation

Translates instructions dynamically into “Uops”

- ~118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine

- Uop executed when
 - Operands available
 - Functional unit available
- Execution controlled by “Reservation Stations”
 - Keeps track of data dependencies between uops
 - Allocates resources

High-Performance Branch Prediction

Critical to Performance

- Typically 11–15 cycle penalty for misprediction

Branch Target Buffer

- 512 entries
- 4 bits of history
- Adaptive algorithm
 - Can recognize repeated patterns, e.g., alternating taken–not taken

Handling BTB misses

- Detect in ~cycle 6
- Predict taken for negative offset, not taken for positive
 - Loops vs. conditionals

Branching Structures

Predict not taken works well for “top of the loop” branching structures

- But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead

```
Loop:  cml %eax, %edx
       je Out
       1nd loop instr
       .
       .
       last loop instr
       jmp Loop
Out:    fall out instr
```

Predict not taken doesn't work well for “bottom of the loop” branching structures

```
Loop:  1st loop instr
       2nd loop instr
       .
       .
       last loop instr
       cml %eax, %edx
       jne Loop
       fall out instr
```

Branch Prediction Algorithms

Static Branch Prediction

- Prediction (taken/not-taken) either assumed or encoded into program

Dynamic Branch Prediction

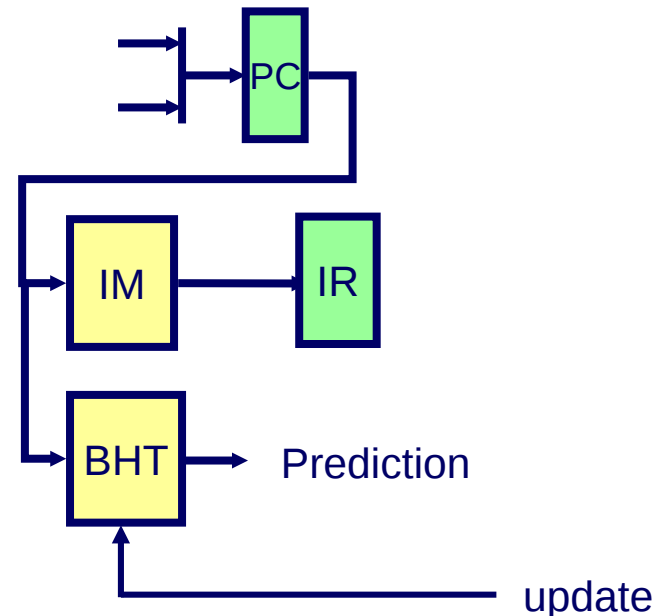
- Uses forms of machine learning (in hardware) to predict branches
 - Track branch behavior
- Past history of individual branches
- Learn branch biases
- Learn patterns and correlations between different branches
- Can be very accurate (95% plus) as compared to less than 90% for static

Simple Dynamic Predictor

Predict branch based on past history of branch

Branch history table

- Indexed by PC (or fraction of it)
- Each entry stores last direction that indexed branch went (1 bit to encode taken/not-taken)
- Table is a cache of recent branches
- Buffer size of 4096 entries are common (track 4K different branches)



Multi-bit predictors

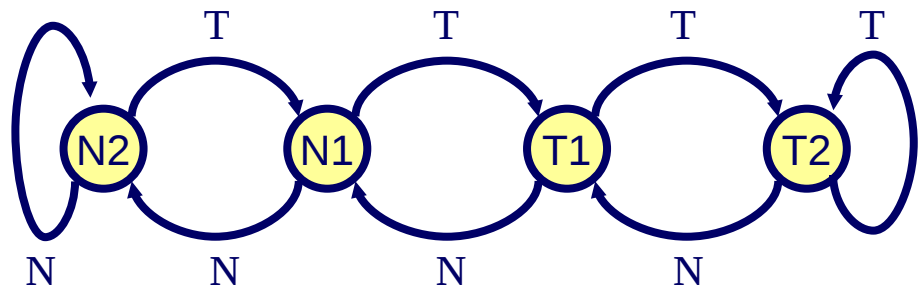
A 'predict same as last' strategy gets two mispredicts on each loop

- Predict NTTT...TTT
- Actual TTTT...TTN

Can do much better by adding *inertia* to the predictor

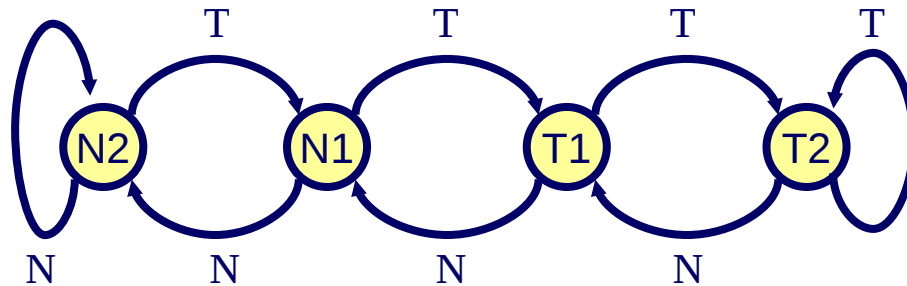
- e.g., two-bit saturating counter
- Predict TTTT...TTT
- Use two bits to encode:
 - Strongly taken (T2)
 - Weakly taken (T1)
 - Weakly not-taken (N1)
 - Strongly not-taken (N2)

```
for(j=0;j<30;j++) {  
    ...  
}
```



State diagram to representing states and transitions

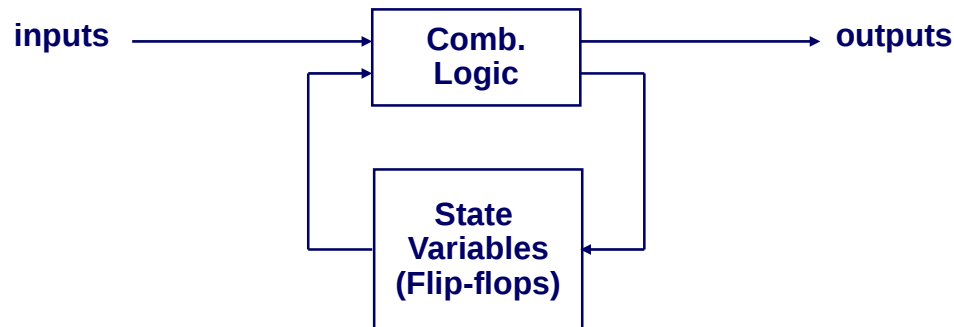
How do we build this in Hardware?



This is a sequential logic circuit that can be formulated as a state machine

- 4 states (N2, N1, T1, T2)
- Transitions between the states based on action “b”

General form of state machine:



State Machine for Branch Predictor

4 states - can encode in two state bits $\langle S1, S0 \rangle$

- $N2 = 00, N1 = 01, T1 = 10, T2 = 11$
- Thus we only need 2 storage bits (flip-flops in last slide)

Input: $b = 1$ if last branch was taken, 0 if not taken

Output: $p = 1$ if predict taken, 0 if predict not taken

Now - we just need combinational logic equations for:

- $p, S1_{\text{new}}, S0_{\text{new}}$, based on $b, S1, S0$

Combinational logic for state machine

$p = 1$ if state is T2 or T1

thus $p = S1$ (according to encodings)

The state variables $S1$, $S0$ are governed by the truth table that implements the state diagram

- $S1_{new} = S1*S0 + S1*b + S0*b$
- $S0_{new} = S1*S0' + S0'*S1'*b + S0*S1*b$

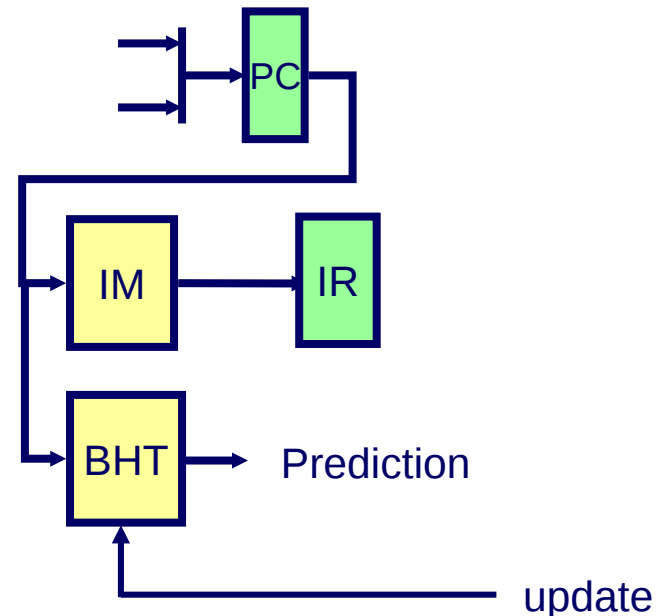
S1	S0	b	S1 _{new}	S0 _{new}	p
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	1	1

Enhanced Dynamic Predictor

Replace simple table of 1 bit histories with table of 2 bit state bits

State transition logic can be shared across all entries in table

- Read entry out
- Apply combinational logic
- Write updated state bits back into table



YMSBP

Yet more sophisticated branch predictors

Predictors that recognize patterns

- eg. if last three instances of a given branches were NTN, then predict taken

Predictors that correlate between multiple branches

- eg. if the last three instances of any branch were NTN, then predict taken

Predictors that correlate weight different past branches differently

- e.g. if the branches 1, 4, and 8 ago were NTN, then predict taken

Hybrid predictors that are composed of multiple different predictors

- e.g. two different predictors run in parallel and a third predictor predicts which one to use

More sophisticated learning algorithms

Branch target buffers

Predictor tells us taken/not-taken

- Actual target address still must be calculated

Branch target buffer contains the predicted target address

- Allows speculative fetch to occur earlier in pipeline
- Requires more storage (PC, not just prediction state)

Summary

Today

- Branch mispredictions cost a lot in performance
- CPU Designers willing to go to great lengths to improve prediction accuracy
- Predictors are just state machines that can be designed using combinational logic and flip-flops