CPU Performance Equation

$$Time\ for\ task = C * T * I$$

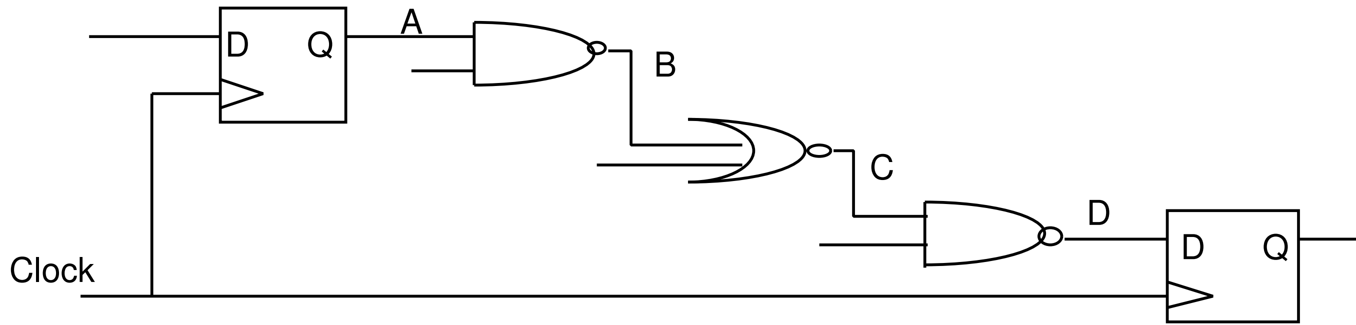
C =Average # Cycles per instruction
T =Time per cycle
I =Instructions per task

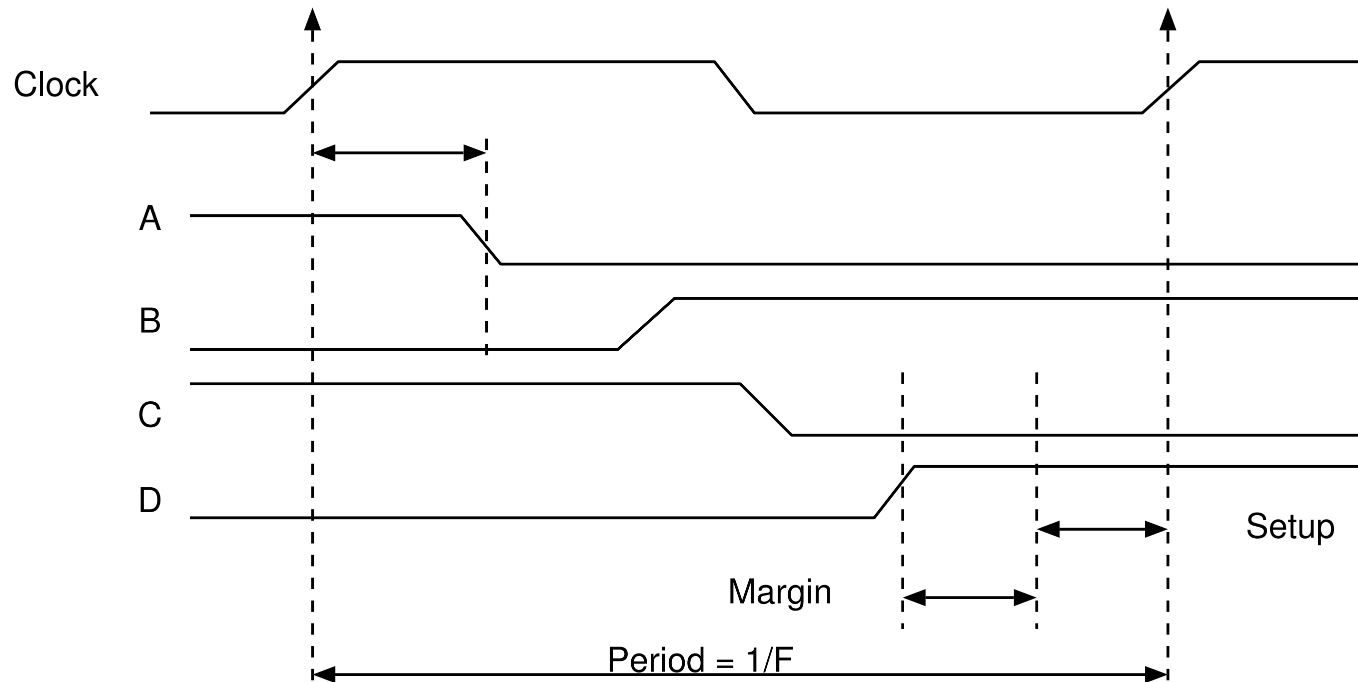
- Pipelining
  - e.g. 3-5 pipeline steps (ARM, SA, R3000)
  - Attempt to get C down to 1
  - Problem: stalls due to control/data hazards
- Super-Pipelining
  - e.g. 8+ pipeline steps (R4000)
  - Attempt to decrease T

- Problem: stalls harder to avoid

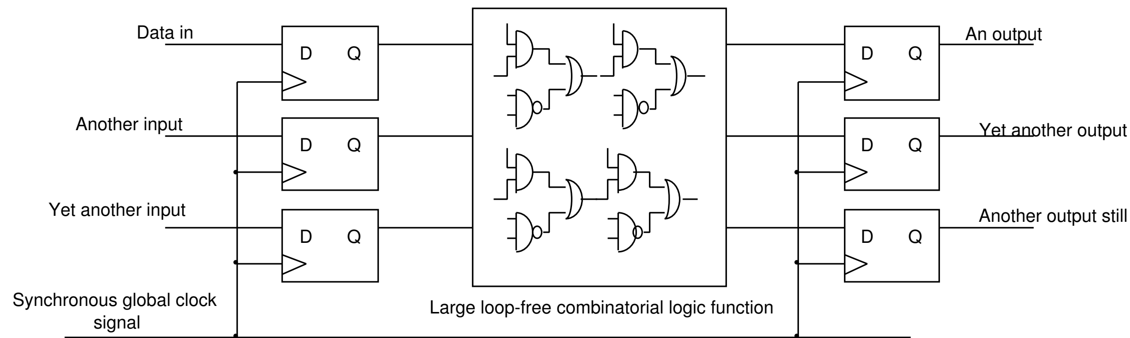
- Super-Scalar
  - Issue multiple instructions per clock
  - Attempt to get C below 1
  - Problem: finding parallelism to exploit
    - typically Instruction Level Parallelism (ILP)

Critical Path

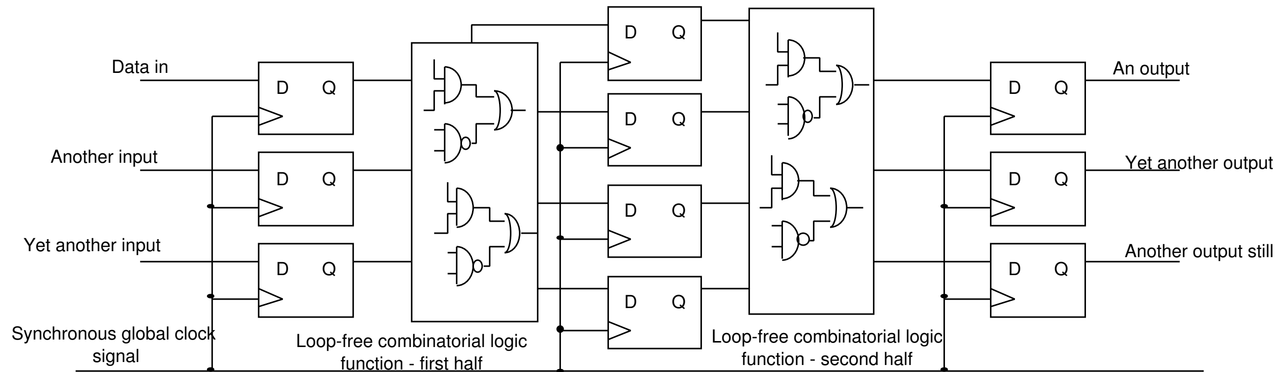




Pipelining

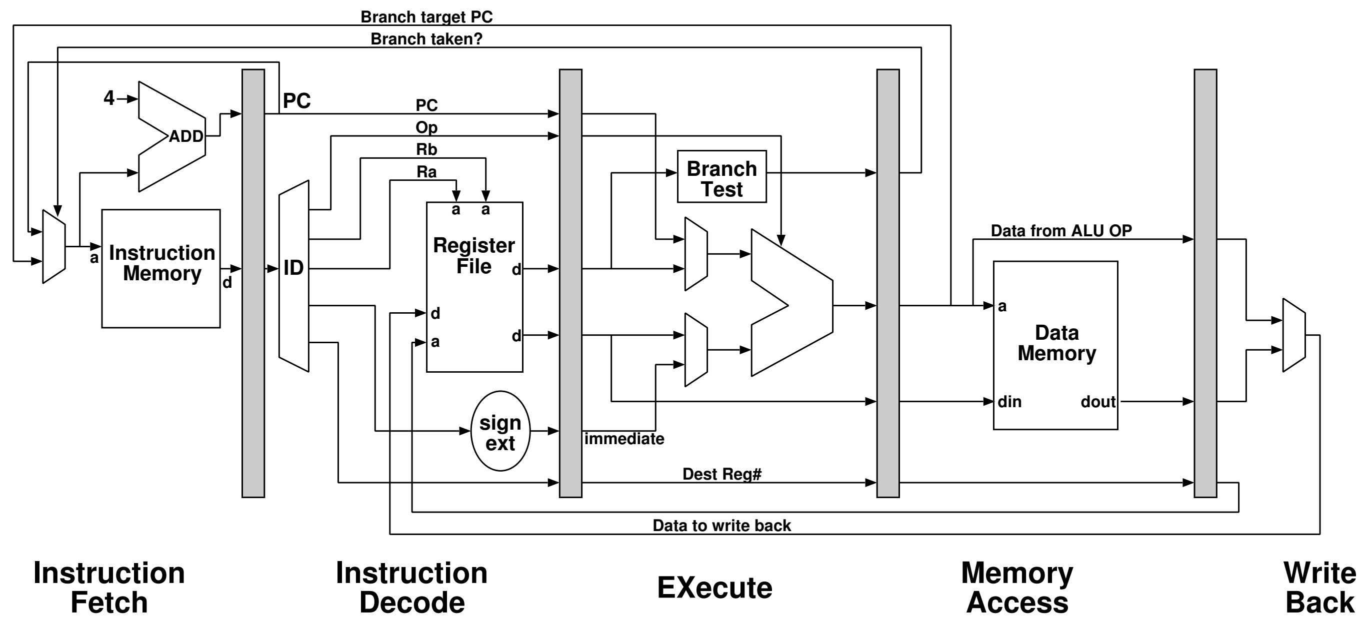


Desired logic function



Desired logic function - pipelined version.

The classic RISC pipe



IF	Send out PC to I-cache. Read instruction into IR. Increment PC.
ID	Decode instruction and read registers in parallel (possible because of fixed instruction format). Sign extend any immediate value.
EX	Calculate Effective Address for Load/Stores. Perform ALU op for data processing instructions. Calculate branch address. Evaluate condition to decide whether branch taken.
MA	Access memory if load/store.
WB	Write back load data or ALU result to register file.

The cost of pipelining

- Pipeline latches add to cycle time

- Cycle time determined by slowest stage
  - Try to balance each stage

- Some resources need to be duplicated to avoid some Structural Hazards
  - (PC incrementer)
  - Multiple register ports (2R/1W)
  - Separate I&D caches

$\Rightarrow$ Effectiveness determined by CPI achieved

Pipelining is efficient

Non Load-Store Architectures

- Long pipe with multiple add and memory access stages
  - Lots of logic
  - Many stages unused by most instructions
- Or, multiple passes per instruction
  - Tricky control logic
- Or, convert architectural instructions into multiple RISC-like internal operations
  - Good for multi-issue
  - More ID stages
  - Pentium Pro/II/III (μops)
  - AMD x86 K7 (r-ops)

Pipelining easiest if all instructions do a similar amount of 'work'

Data Hazard Taxonomy

- WaW: write after write, out of order completion.
- WaR: write too early, overwrites data still needed.
- RaW: need to read a value that has (potentially) yet to be written.

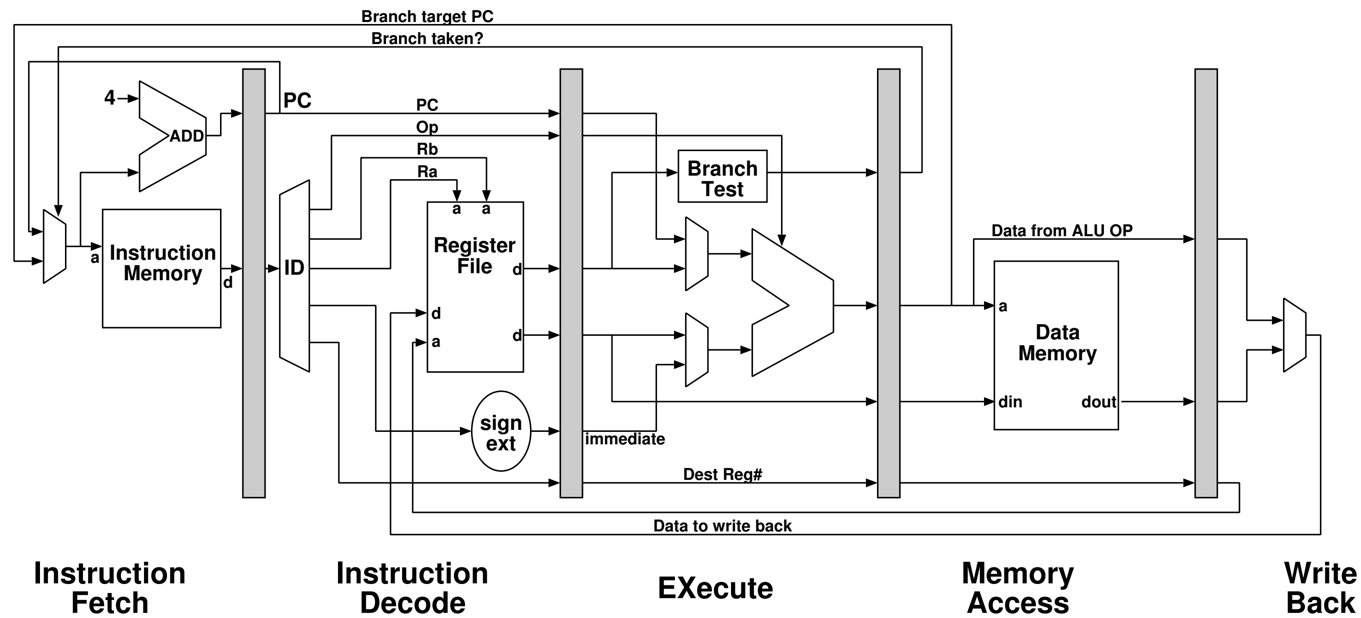
The first two are avoided when have sufficient registers and/or can rename.

RaW hazards are more tricky!

ALU Result Forwarding

- E.g. 4 forwarding paths to avoid stalls:

```
a:  add r1 ← r8, r9
b:  add r2 ← r1, r7
c:  add r3 ← r1, r2
d:  add r4 ← r1, r2
```



- Read after Write
- Doubled # mux inputs
- Deeper pipes $\rightarrow$ more forwarding
  - R4000, 8 deep pipe
    forward to next 4 instructions

Load Data Hazards

- Impossible without a stall:

```
lw r1 ← r9(4)

add r2 ← r1, r6
```

- Read after Write (RaW) hazard

- New forwarding path saves a cycle

- Re-order code to avoid stall cycle
  - Possible for 60-90% of loads

- Software Interlocked
  - Compiler must insert `nop`

- e.g. R2000/R3000

- Hardware Interlocked
  - Save `nop`: better for I-stream density
  - Register scoreboard
    * track location of reg values e.g.:
    * File, EX, MA, MUL1, MUL2, MemVoid
    * hold back issue until RaW hazard resolved
    * control operand routeing
  - Required for all sophisticated pipelines

- More stalls for deeper pipes
  - 2 stalls and 2 more forwarding paths for R4000

Longer Latency Instructions

- Mul/Div, Floating Point

- Different functional units operating in parallel with main pipeline

- Extra problems:
  - Structural hazards
    - Unit may not be fully pipelined, eg:
      - 21264 FDiv: 16cy latency, not pipelined
      - 21164 FMul: 8cy latency, issue every 4cy

      $\rightarrow$ 21264 FMul: 4cy latency, fully pipelined

    - Multiple Write Back stages
        - more register write ports?
        - or, schedule pipeline bubble
  - Read after Write hazards more likely
    - compiler instruction scheduling
  - Instruction complete out of order
    - Write after Write hazards possible
    - Dealing with interrupts/exceptions

- Use scoreboard to determine when safe to issue

- Often hard to insert stalls after ID stage
  - synthesize NOPs in ID to create bubble
  - 'replay trap' : junk & refetch

Exceptions and Pipelining

User SWI/trap	ID	Precise (easy)
Illegal Instruction	ID	Precise (easy)
MMU TLB miss	IF/MA	Precise required
Unaligned Access	MA	Precise required
Arithmetic	EX 1..N	Imprecise possible

- Exceptions detected past the point of in-order execution can be tricky
  - FP overflow
  - Int overflow from Mul/Div
- Exact arithmetic exceptions
  - Appears to stop on faulting instruction
  - Need exact PC
    - ∗ care with branch delay slots

- Roll back state/In-order commit (PPro)

- Imprecise arithmetic exceptions
  - Exception raised many cycles later
  - Alpha: Trap Barriers
  - PPC: Serialise mode
  - IA-64: Poison (NaT) bits on registers
    - instructions propagate poison
    - explicit collection with 'branch if poison'

Interrupts

- Interrupts are asynchronous
- Need bounded latency
  - Real-time applications
  - Shadow registers avoid spilling state
    - Alpha, ARM
- Some CISC instructions may need to be interruptible
  - Resume *vs.* Restart
    - eg. overlapping memcpy
  - Update operands to reflect progress
    - VAX MOVC

Control Flow Instructions

- Absolute *jumps*
  - To an absolute address
    (usually calculated by linker)
  - Immediate / Register modes
  - usage: function pointers, procedure call/return into other compilation modules, shared libraries, switch/case statements
  - Unconditional - but perhaps computed
  - Also subroutine return
- PC Relative *branches*
  - Signed immediate offset
  - Limited range on RISC
    - Typically same compilation module (calculated by compiler)
  - Conditional

Jump to Subroutine often has hardware stack support.

– Save PC of following instruction into:

  * CISC: stack
  * most RISC: special register
  * ALPHA: nominated register
  * IA-64: nominated Branch Reg

ARM neatly optimised RLA save along with data register saving.

Modern CPUs want to know which registers are (likely) to be holding an RLA.

Conditional Branches

- Conditional branch types
  - most: Test condition code flags
    - Z, C, N, V
    - `Bxx label`
  - Alpha/MIPS: Test bits in named reg
    - msb (sign), lsb, ‘zero’
    - `Bxx Ra, label`
  - some: Compare two registers and branch
    - `Bxx Ra, Rb, label`
    - (PA-RISC, MIPS, some CISC)

  - IA-64: Test one of 64 single bit predicate regs

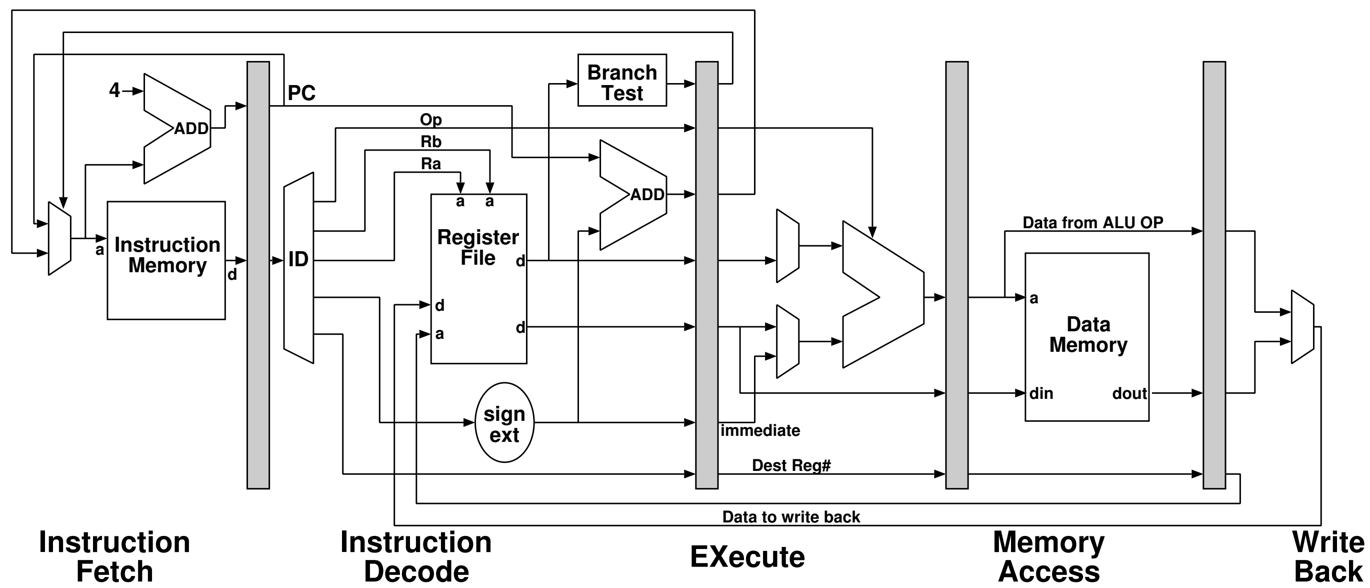
- Conditional branch stats (for MIPS and SPEC92)
  - 15% of executed instructions
    * 73% forward (if/else)
    * 27% backward (loops)
  - 67% of branches are taken
    * 60% forward taken
    * 85% backward taken (loops)

Control Hazards

- 'classic' evaluates conditional branches in EX
  - Identify branch in ID, and stall until outcome known
  - Or better, assume not taken and abort if wrong

  $\rightarrow$ 2 stall taken-branch penalty

- If evaluating branch is simple, replicate h/w to allow early decision
  - Branch on condition code
  - Alpha/MIPS: Test bits in named reg
    - Special 'zero' bit stored with each reg
  - Hard if `Bxx Ra, Rb, label`



4
ADD
PC
Op
Rb
Ra
Branch Test
ADD
Instruction Memory
a
d
ID
Register File
a
a
d
d
d
a
sign ext
immediate
Dest Reg#
Data from ALU OP
Data Memory
a
din
dout
Data to write back
Instruction Fetch
Instruction Decode
EXecute
Memory Access
Write Back

Control Hazards (2)

- Evaluate branches in ID (when possible)

⇒ Only 1 cycle stall if test value ready
(Set flags/reg well before branch)

  - Bad if every instruction sets flags (CISC)
  - Helps if setting CC optional (SPARC/ARM)
  - Good if multiple CC regs (PPC/IA-64), or none (Alpha/MIPS)

- Branch delay slots avoided the taken branch stall on early MIPS

  - Always execute following instruction

  - Can't be another branch
  - Compiler fills slot ~60% of the time
  - Branches with optional slots: avoid `nop`

- Modern CPUs typically have more stages before EX, due to complicated issue-control logic, thus implying a greater taken-branch cost

- Stalls hurt more on a multi-issue machine. Also, fewer cycles between branch instructions

Control hazards can cripple multi-issue CPUs

Static Branch Prediction

- Speculation should not change semantics!
- Simple prediction
  - e.g. predict backward as taken, forward not
- Branch instructions with hints
  - Branch likely/unlikely
    - strong/weak hint varients
  - Use Feedback Directed Optimization (FDO)
  - Fetch I-stream based on hint
- Delayed branch instrs with hints and annulment
  - If hint that branch taken is correct then execute slot instruction else don't
  - e.g. new MIPS, PA-RISC
  - Compiler able to fill delay slot more easily

Dynamic Branch Prediction

- Static hints help, but need to do better

- Branch prediction caches
  - Indexed by significant low order bits of branch instruction address
  - Cache entries do not need tags (they're only hints)
  - E.g. 512-8K entries

- Bi-modal prediction method

  $\Rightarrow$ many branches are strongly biased

  - Single bit predictor

  * Bit predicts branch as taken/not taken
  * Update bit with actual behaviour
  * Gets first and last iterations of loops wrong
- Two bit predictors
  * Counter saturates at 0 and 3
  * Predict taken if 2 or 3
  * Add 1 if taken, subtract 1 if not
  * Little improvement above two bits
  * $\geq$90% for 4K entry buffer on SPEC92

Local History predictors

- Able to spot repetitive patterns
- Copes well with minor deviations from pattern
- E.g. 4 bit local history branch predictor
  - 4 bit shift reg stores branch's prior behaviour
  - 16 x 2 bit bi-modal predictors per entry
  - use shift reg to select predictor to use
  - perfectly predicts all patterns $<$ length 6, as well as some longer ones (up to length 16)
  - used on Pentium Pro / Pentium II

      - ∗ 512 entries x (16 x 2 + 4) = 18K bits SRAM
    - trained after two sequence reps
    - other seqs up to 6% worse than random

- An alternative approach is to use two arrays. One holds branch history, the other is a shared array of counters indexed by branch history
    - branches with same pattern share entries in 2nd array (more efficient)
    - 21264 LH predictor: 1024 entries x 10 bits of history per branch, and shared array of 1024 counters indexed by history

Global Correlating predictors

- Behaviour of some branches is best predicted by observing behaviour of other branches

- (Spatial locality)

$\Rightarrow$ Keep a short history of the direction that the last few branch instructions executed have taken

- E.g. Two bit correlating predictor:
  - 2 bit shift register to hold *processor* branch history
  - 4 bi-modal counters in each cache entry, one for each possible global history

- Rather than using branch address, some GC predictors use the processor history as the *index* into a single bi-modal counter array. Also possible to use a hash of (branch address, global history)
  - Alpha 21264 GC predictor uses a 12 bit history and 4096 x 2 bit counters

- Combination of Local History and Global Correlating predictor works well
  - $\geq$95% for 30K bit table on SPEC92
  - E.g. Alpha 21264
  - Tournament System to decide which to use.

Reducing Taken-Branch Penalty

- Branch predictors usually accessed in ID stage, hence at least one bubble required for taken-branches

- Need other mechanisms to try and maintain a full stream of useful instructions:

- Branch target buffers
  - In parallel with IF, look up PC in BTB
  - if PC is present in BTB, start fetching from the address indicated in the entry
  - Some BTBs actually cache instructions from the target address

- Next-fetch predictors
  - Very simple, early, prediction to avoid fetch bubbles, used on PPro, A21264
  - I-cache lines have pointer to the next line to fetch
  - Update I-cache ptr. based on actual outcome
- Trace caches (Pentium IV)
  - Replace traditional I-cache
  - Cache commonly executed instr sequences, crossing basic block boundaires
  - (c.f. “trace straightening” s/w optimization)
  - Table to map instr address to position in cache
  - Instrs typically stored in decoded form

Avoiding branches

- Loop Count Register (PowerPC, x86, IA-64)
  - Decrement and branch instruction
  - Only available for innermost loops

- Predicated Execution (ARM, IA-64)
  - Execution of all instructions is conditional
    - ARM: on flags registers
    - IA-64: nominated predicate bit (of 64)
  - IA-64: cmp instructions nominate two predicate bits, one is set and cleared depending on outcome
  - E.g. if([r1] && [r2] && [r3]) {...} else {...}

```
ld r4 <- [r1]
p6,p0 <= cmp( true )
p1,p2 <= cmp( r4==true )
<p1> ld r5 <- [r2]
<p1> p3,p4 <= cmp( r5==true )
<p3> ld r6 <- [r3]
<p3> p5,p6 <= cmp( r6==true )
<p6> br else
...
```

- ✔ Transform control dependency into data dep
- ✔ Instruction ‘boosting’
  - ∗ e.g. hoist a store ahead of a branch
- ✔ Inline simple if/else statements
- ✘ Costs opcode bits
- ✘ Issue slots wasted executing nullified instrs

Avoiding branches 2

- Conditional Moves (Alpha, new on MIPS and x86)
  - Move if flags/nominated reg set
  - Just provides a 'commit' operation
    - beware side effects on 'wrong' path
  - PA-RISC supports arbitrary nullification

- Parallel Compares (IA-64)
  - Eliminate branches in complex predicates
  - Evaluate in parallel
    - (despite predicate dependancy)

– if ((rA<0) && (rB==-15) && (rC>0)) {...}

```
cmp.eq p1,p0 = r0, r0 ;; // p1 =1
cmp.ge.and p1,p0 = rA,r0
cmp.ne.and p1,p0 = rB,-15
cmp.le.and p1,p2 = rB,10
(p1) br.cond if-block
```

Avoid hard to predict branches

Optimizing Jumps

- Alpha: Jumps have static target address hint
  - A_{16-2} of target instruction virtual address
  - Enough for speculative I-cache fetch
  - Filled in by linker or dynamic optimizer
- Subroutine Call Returns
  - Return address stack
  - Alpha: Push/pop hints for jumps
  - 8 entry stack gives $\geq$ 95% for SPEC92
- Jump target registers (PowerPC/IA64)
  - Make likely jump destinations explicit
  - Buffer instructions from each target
- Next-fetch predictors / BTBs / trace caches help for jumps too
  - Learn last target address of jumps
  - Good for shared library linkage