# Change3-4
# Multiple-issue
# ----SuperScalar
# & VLIW

# Review

## – explore ILP via Hardware approaches

- Basic 5-stage pipeline
- Extended to pipeline supporting FP operations
- Scoreboard
- Tomasulo Algorithm
- Branch predictor
- Hardware-based Speculation
- Explicit register renaming

# Review: achieve an ideal CPI = 1 !

- Forwarding
  - Reduce potential data hazard stalls.
- Delayed branch
  - Reduce control stalls with simple branch scheduling
- Dynamic scheduling
  - Scoreboard:
    - reduce data stalls from true data dependence
  - Tomasulo:
    - Eliminate data stalls from WAR & WAW data dependences via renaming
    - Reduce data hazard stalls via out-of-order execution
- Branch predictor
  - Reduce control stalls
- Hardware-based speculation
  - Branch predictor + dynamic scheduling + speculation

# Getting CPI < 1
# Multiple Issue Processors:

- Vector Processing: Explicit coding of independent loops as operations on large vectors of numbers
  - Multimedia instructions being added to many processors
- Superscalar: varying number of instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium III/4
- (Very) Long Instruction Words (V)LIW: fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates (TBD)
  - Intel Architecture-64 (IA-64) 64-bit address
    - Renamed: "Explicitly Parallel Instruction Computer (EPIC)"
- Anticipated success of multiple instructions lead to Instructions Per Clock_cycle (IPC) vs. CPI

# Explore ILP via Multiple-issue

- Goal:
  - Allow multiple instructions to issue in a clock cycle. Getting CPI < 1:
- Approach
  - Static Superscalar
  - Dynamic Superscalar
  - Speculative Superscalar
  - VLIW/LIW
  - EPIC( IA-64)

# Comparison

| | Issue Structure | Hazard Detection | Scheduling | Characteristic | examples |
|---|---|---|---|---|---|
| Superscalar (static) | Dynamic | Hardware | Static | In-order execution | Sun UltraSPARCII |
| Superscalar (dynamic) | Dynamic | Hardware | Dynamic | Some out-or-order exec. | IBM Power2 |
| Superscalar (speculative) | Dynamic | Hardware | Dynamic With speculation | Out-of-order exec. With speculation | Pentium III/4 MIPS R10K, Alpha 21264 |
| VLIW/ LIW | Static | Software | Static | No hazards in issue packets | Trimedia, i860 |
| EPIC | Mostly static | Mostly software | Mostly static | Explicit dependences marked by compiler | Itanium |

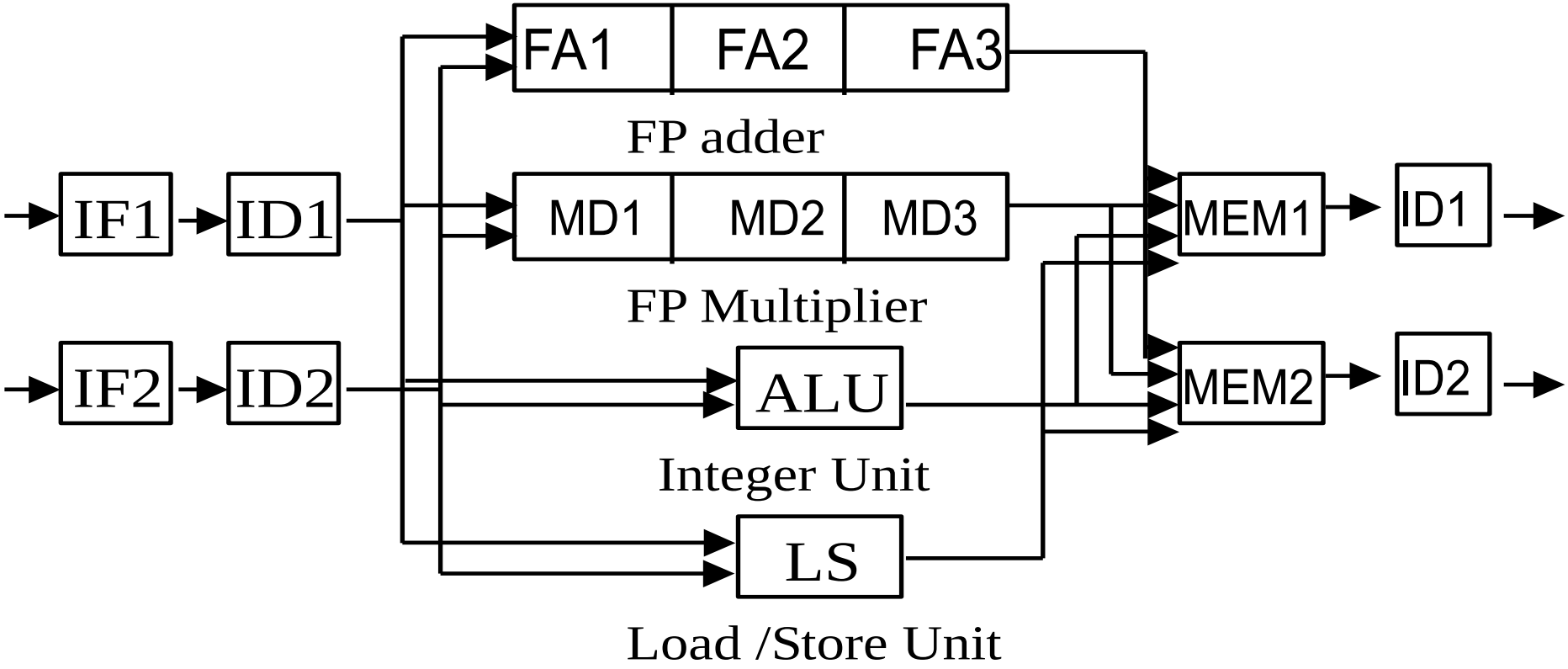
# Superscalar

- **the processor tries to issue more than one instruction (varying number 1-8 ) per cycle so as to keep all of the functional units busy.**
  - Statically scheduled
    - using compiler techniques
    - In-order execution
  - Dynamically scheduled
    - using techniques based on Tomasulo's algorithm
    - Out-of-order execution

# Statically Scheduled Superscalar

- Instruction issue in order
- All pipeline hazards are checked for at issue time. May issue 0~8 instructions per Clockcycle.
- Issue packet: the instruction group received from the fetch unit that potentially issue in one clock cycle.
- Issue stage is split and pipelined:
  - Decide how many instructions from the packet can issue simultaneously (within packet )
  - Detect hazards among the selected instructions and those that have already been issued.(between packet)

# An example of dual-issue pipeline



# Ex. Superscalar MIPS

- 2 instructions, 1 FP & 1 anything
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair
- 
- 1 cycle load delay expands to 3 instructions in Superscalar
  - instruction in right half can't use it, nor instructions in next slot

- Branch delay for a taken branch becomes either two or three instructions

# Superscalar MIPS pipeline in operation

| Inst. type | Pipe stages | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Int. instruction | IF | ID | EX | MEM | WB | | | | |
| FP instruction | IF | ID | EX | MEM | WB | | | | |
| Int. instruction | | IF | ID | EX | MEM | WB | | | |
| FP instruction | | IF | ID | EX | MEM | WB | | | |
| Int. instruction | | | IF | ID | EX | MEM | WB | | |
| FP instruction | | | IF | ID | EX | MEM | WB | | |
| Int. instruction | | | | IF | ID | EX | MEM | WB | |
| FP instruction | | | | IF | ID | EX | MEM | WB | |

# Multiple Issues

- issue packet: group of instructions from fetch unit that could potentially issue in 1 clock
  - If instruction causes structural hazard or a data hazard either due to earlier instruction in execution or to earlier instruction in issue packet, then instruction does not issue
  - 0 to N instruction issues per clock cycle, for N-issue
- Performing issue checks in 1 cycle could limit clock cycle time: $O(n^2-n)$ comparisons
  - => issue stage usually split and pipelined
  - 1st stage decides how many instructions from within this packet can issue, 2nd stage examines hazards among selected instructions and those already been issued
  - => higher branch penalties => prediction accuracy important

# Multiple Issue Challenges

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations AND No hazards
- If more instructions issue at same time, greater difficulty of decode and issue:
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue; (N-issue ~O($N^2$-N) comparisons)
  - Register file: need 2*N reads and 1*N writes/cycle

# Multiple Issue Challenges(cont.)

- Rename logic: must be able to rename same register multiple times in one cycle! For instance, consider 4-way issue:

```
add r1, r2, r3                 add p11, p4, p7
sub r4, r1, r2        ⇒        sub p22, p11, p4
lw  r1, 4(r4)                  lw  p23, 4(p22)
add r5, r1, r2                 add p12, p23, p4
```

  Imagine doing this transformation in a single cycle!

- Result buses: Need to complete multiple instructions/cycle
  - So, need **multiple buses** with associated matching logic at every reservation station.
  - Or, need **multiple forwarding paths**

# Dynamically Scheduled Superscalar

- Potentially overcome the issue restrictions.
- Two different approaches to issue multiple instructions per clock:
  - Pipeline: Run this step in half a clock cycle, so that two instructions can be processed in one clock cycle.
  - Widen issue logic: Build the logic necessary to handle two instructions at once, including any possible dependences between the instructions.
  - Both: Modern superscalar processors often include both pipeline and widen the issue logic.

# Example

```
Loop:  L.D      F0, 0(R1)
       ADD.D    F4, F0, F2
       S.D      F4, 0(R1)
       DADDIU   R1, R1, #-8
       BNE      R1,R2, Loop
```

**Function Latency:**

- **1 cycle for integer ALU**
- **2 cycles for load**
- **3 cycles for FP add.**

Assumptions:

- 1 FP and 1 integer operation per CC. even if they are dependent.
- One interger function unit for ALU and address calculations
- Separate function unit for evaluating branch condition
- Single issue for branches, but perfect prediction.
- No speculation: all instr. following a branch are delay until branch resolved.

# Operation on a dual-issue version of Tomasulo pipeline

**One memory unit, one interger pipeline, one FP adder**

| Iteration number | Instructions | Issues at | Executes | Memory Access at | Write CDB at |
|---|---|---|---|---|---|
| 1 | LD F0, 0(R1) | 1 | 2 | 3 | 4 |
| 1 | ADDD F4, F0, F2 | 1 | 5 | | 8 |
| 1 | SD 0(R1), F4 | 2 | 3 | 9 | |
| 1 | SUBI R1, R1, #-8 | 2 | 4 | | 5 |
| 1 | BNEZ R1, LOOP | 3 | 6 | | |
| 2 | LD F0, 0(R1) | 4 | 7 | 8 | 9 |
| 2 | ADDD F4, F0, F2 | 4 | 10 | | 13 |
| 2 | SD 0(R1), F4 | 5 | 8 | 14 | |
| 2 | SUBI R1, R1, #8 | 5 | 9 | | 10 |
| 2 | BNEZ R1, LOOP | 6 | 11 | | |

**Structure hazard**

**No speculation**

**Structure hazard**

**Structure hazard**

**Integer function unit → bottleneck**

# Separate FU for ALU op and Address Calculation

**A second CDB is needed !**

| Iteration number | Instructions | Issues at | Executes | Memory Access at | Write CDB at |
|---|---|---|---|---|---|
| 1 | LD F0, 0(R1) | 1 | 2 | 3 | 4 |
| 1 | ADDD F4, F0, F2 | 1 | 5 | | 8 |
| 1 | SD 0(R1), F4 | 2 | 3 | 9 | |
| 1 | SUBI R1, R1, #-8 | 2 | 3(4) | | 4 |
| 1 | BNEZ R1,R2, LOOP | 3 | 5 | | |
| 2 | LD F0, 0(R1) | 4 | 6 | 7 | 8 |
| 2 | ADDD F4, F0, F2 | 4 | 9 | | 12 |
| 2 | SD 0(R1), F4 | 5 | 7 | 13 | |
| 2 | SUBI R1, R1, #8 | 5 | 6(9) | | 7 |
| 2 | BNEZ R1, R2,LOOP | 6 | 8 | | |

# Multiple Issue with Speculation

- A speculative processor can be extended to multiple issue.
- Need to handle multiple instruction commits per clock cycle.
- Example:

```
Loop:  LD   R2, 0(R1)
       ADDI   R2, R2, #1
       SD     R2, 0(R1)
       ADDI   R1, R1, #4
       BNE    R2, R3, Loop
```

# Assumptions

- Separate integer function units for
  - Effective address calculation
  - ALU operations
  - Branch condition evaluation

# Dual-issue without speculation

| Iteration number | Instructions | Issues at | Executes | Memory Access at | Write CDB at | Comment |
|---|---|---|---|---|---|---|
| 1 | LD R2, 0(R1) | 1 | 2 | 3 | 4 | First issue |
| 1 | ADDI R2, R2, #1 | 1 | 5 | | 6 | Wait foe LW |
| 1 | SD R2, 0(R1) | 2 | 3 | 7 | | Wait for ADDI |
| 1 | ADDI R1, R1, #-4 | 2 | 3 | | 4 | Execute directly |
| 1 | BNE R2, R3, LOOP | 3 | 7 | | | Wait for ADDI |
| 2 | LD R2, 0(R1) | 4 | 8 | 9 | 10 | Wait for BNE |
| 2 | ADDI R2, R2, #1 | 4 | 11 | | 12 | Wait for LW |
| 2 | SD R2, 0(R1) | 5 | 9 | 13 | | Wait for ADDI |
| 2 | ADDI R1, R1, #-4 | 5 | 8 | | 9 | Wait for BNE |
| 2 | BNE R2, R3, LOOP | 6 | 13 | | | Wait for ADDI |

# Dual-issue with speculation

| Iteration number | Instructions | Issues at | Executes | Memory Access at | Write CDB at | Commits at clock number |
|---|---|---|---|---|---|---|
| 1 | LD R2, 0(R1) | 1 | 2 | 3 | 4 | 5 |
| 1 | ADDI R2, R2, #1 | 1 | 5 | | 6 | 7 |
| 1 | SD R2, 0(R1) | 2 | 3 | | | 7 |
| 1 | ADDI R1, R1, #-4 | 2 | 3 | | 4 | 8 |
| 1 | BNE R2, R3, LOOP | 3 | 7 | | | 8 |
| 2 | LD R2, 0(R1) | 4 | 5 | 6 | 7 | 9 |
| 2 | ADDI R2, R2, #1 | 4 | 8 | | 9 | 10 |
| 2 | SD R2, 0(R1) | 5 | 6 | | | 10 |
| 2 | ADDI R1, R1, #-4 | 5 | 6 | | 7 | 11 |
| 2 | BNE R2, R3, LOOP | 6 | 10 | | | 11 |

# Explore ILP via Software approaches

- Basic Compiler Technique for Exposing ILP
  - Loop unrolling
- Static Branch Prediction
- Static multiple Issue: VLIW
- Advanced Compilor Support for Exposing and Exploiting ILP
  - Software pipelining
  - Global Code scheduling
- Hardware Support for Exposing More Parallelism at compile time
  - Conditional or Predicated instructions
  - Compiler speculation with hardware support

# Review: Static Branch Prediction

- Simplest: Predict taken
  - average misprediction rate = untaken branch frequency, which for the SPEC programs is 34%.
  - Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%)
- Predict on the basis of branch direction?
  - choosing backward-going branches to be taken (loop)
  - forward-going branches to be not taken (if)
  - SPEC programs, however, most forward-going branches are taken => predict taken is better
- Predict branches on the basis of profile information collected from earlier runs
  - Misprediction varies from 5% to 22%

# Example:

```
For ( i=1000; i>0; i=i-1 )
        x[i] = x[i] + s;
```

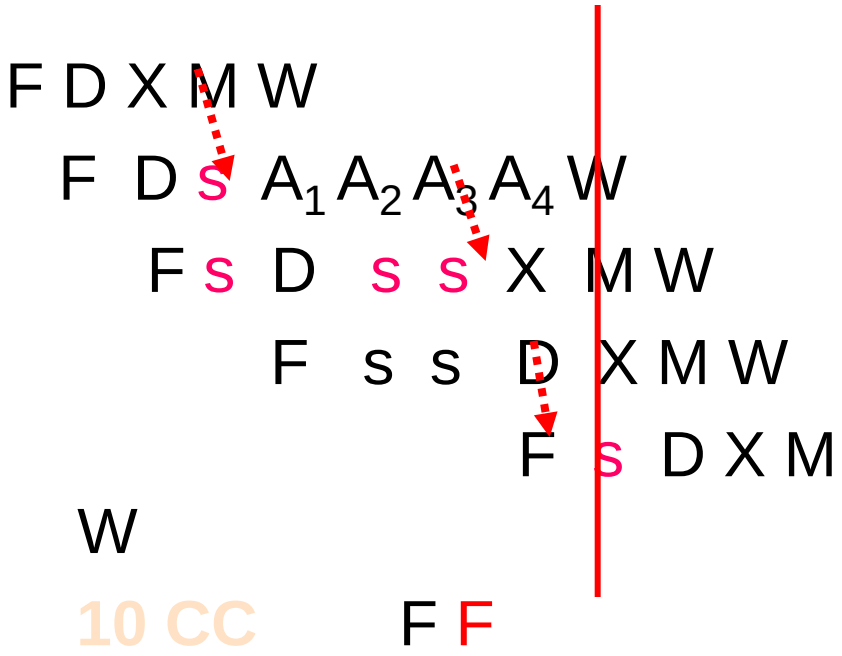
# First: Translate into MIPS code

```
Loop:   LD      F0,0(R1)        ;F0=vector element
        ADDD    F4,F0,F2        ;add scalar from F2
        SD      0(R1),F4        ;store result
        SUBI    R1,R1,8         ;decrement pointer 8B (DW)
        BNEZ    R1,R2,Loop              ;branch if R1 != R2
        NOP                     ;delayed branch slot
```

| *Instruction producing result* | *Instruction using result* | *Latency in cycles* |
|---|---|---|
| **FP ALU op** | **Another FP ALU op** | **3** |
| **FP ALU op** | **Store double** | **2** |
| **Load double** | **FP ALU op** | **1** |
| **Load double** | **Store double** | **0** |
| **Integer op** | **Integer op** | **0** |

# Where are the Hazards?

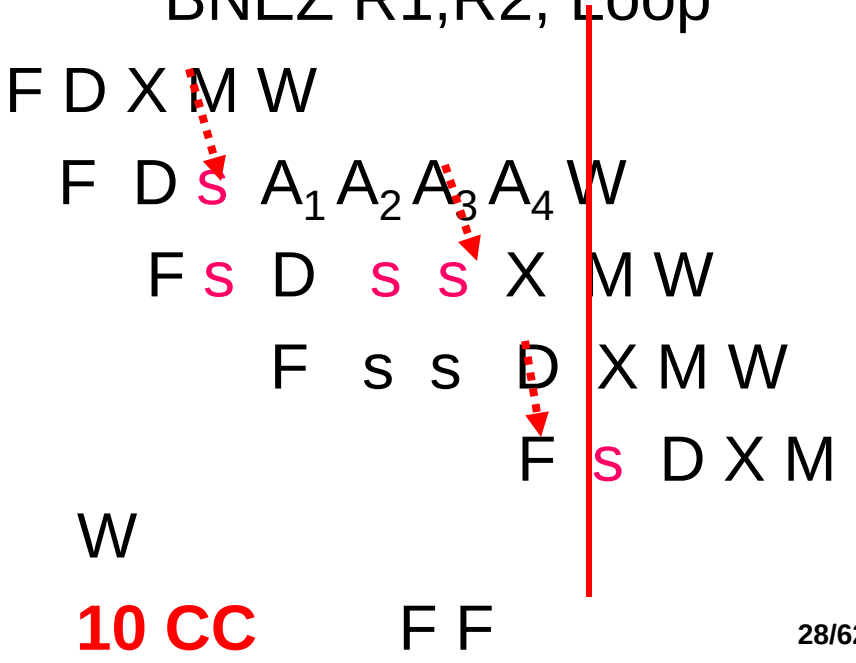
```
Loop: LD   F0, 0(R1)
      ADDD F4, F0, F2
      SD   0(R1),  F4
      SUBI R1, R1, #8
      BNE R1,R2 Loop
```




```
1 Loop: L.D      F0,0(R1)
2  stall
3        ADD.D  F4,F0,F2
4  stall
5  stall
6        S.D      0(R1),F4
7        DSUBUI  R1,R1,8
8  stall
9         BNEZ   R1,R2,Loop
10 stall
```

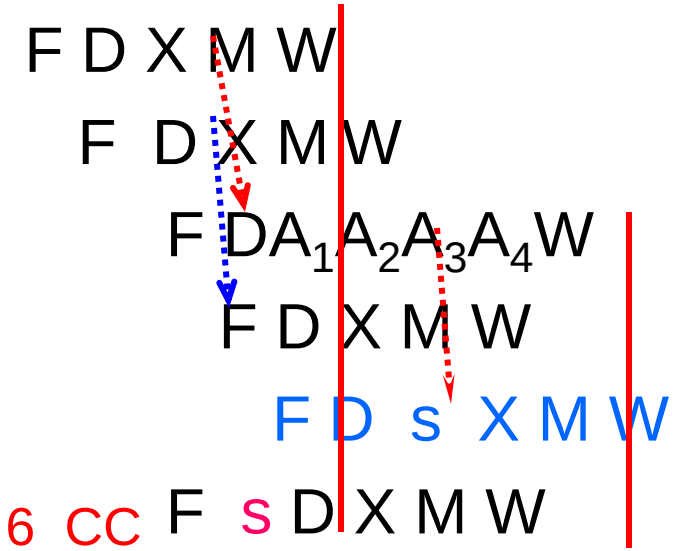
# Reducing stalls from scheduling in Basic and delayed branch

Loop: LD F0, 0(R1)
ADDD F4, F0, F2
SD 0(R1), F4
SUBI R1, R1, #8
BNEZ R1,R2, Loop






**Loop: LD F0, 0(R1)**
**SUBI R1, R1,#8**
**ADDD F4, F0, F2**
**BNEZ R1,R2 Loop**
**SD +8(R1), F4**



# Unrolled Loop That Minimizes Stalls

```
1 Loop:  LD     F0,0(R1)
2        LD     F6,-8(R1)
3        LD     F10,-16(R1)
4        LD     F14,-24(R1)
5        ADDD   F4,F0,F2
6        ADDD   F8,F6,F2
7        ADDD   F12,F10,F2
8        ADDD   F16,F14,F2
9        SD     0(R1),F4
10       SD     -8(R1),F8
11       SUBI   R1,R1,#32
12       SD     +16(R1),F12
13       BNEZ   R1,LOOP
14       SD     8(R1),F16       ; 8-32 = -24
```

- What assumptions made when moved code?
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

***14 clock cycles, or 3.5 per iteration***

# Compiler Perspectives on Code Movement

- Compiler concerned about dependencies in program
- Whether or not a HW hazard depends on pipeline
- Try to schedule to avoid hazards that cause performance losses
- (True) Data dependencies (RAW if a hazard for HW)
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- If dependent, can't execute in parallel
- Easy to determine for registers (fixed names)
- Hard for memory ("memory disambiguation") problem:
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

# Compiler Perspectives on Code Movement

- Our example required compiler to know that if R1 doesn't change then:

  0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)

  There were no dependencies between some loads and stores so they could be moved by each other

# Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
  - For large values of n, most of the execution time will be spent in the unrolled loop

# Steps Compiler Performed to Unroll

- Check OK to move the S.D after DSUBUI and BNEZ, and find amount to adjust S.D offset
- Determine unrolling the loop would be useful by finding that the loop iterations were independent
- Rename registers to avoid name dependencies
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
  - requires analyzing memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield same result as the original code

# Not good enough due to limitations

- Amount of overhead amortized with each unroll
  - Overhead: 2/6 → 2/14=1/ 7 → 2/26=1/13

    2/iteration 2/4 iteration 2/8 iteration
- Result in growth of code size
- Potential shortfall in registers.
- What about branch but not loop ?

# Using Loop unrolling and scheduling with static Multiple Issue

| Integer Instruction | FP instruction | Clock cycle |
|---|---|---|
| Loop: L.D F0, 0(R1) | | 1 |
| L.D F0, -8(R1) | | 2 |
| L.D F0, -16(R1) | ADD.D F4, F0. F2 | 3 |
| L.D F0, -24(R1) | ADD.D F8, F6. F2 | 4 |
| L.D F0, -32(R1) | ADD.D F12, F10. F2 | 5 |
| S.D F4, 0(R1) | ADD.D F16, F14. F2 | 6 |
| S.D F8, -8(R1) | ADD.D F20, F18. F2 | 7 |
| S.D F12,-16(R1) | | 8 |
| DADDUI R1, R1, #-40 | | 9 |
| S.D F16, 16(R1) | | 10 |
| BNE R1, R2, Loop | | 11 |
| S.D F20, 8(R1) | | 12 |

# Static Multiple issue: VLIW

- VLIW: Very Long Instruction Word
- Each “instruction” has explicit coding for multiple operations
  - In EPIC, grouping called a “packet”
  - In Transmeta, grouping called a “molecule” (with “atoms” as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

# Loop Unrolling in VLIW

| *Memory reference 1* | *Memory reference 2* | *FP operation 1* | *FP op. 2* | *Int. op/ branch* | *Clock* |
|---|---|---|---|---|---|
| LD F0,0(R1) | LD F6,-8(R1) | | | | 1 |
| LD F10,-16(R1) | LD F14,-24(R1) | | | | 2 |
| LD F18,-32(R1) | LD F22,-40(R1) | ADDD F4,F0,F2 | ADDD F8,F6,F2 | | 3 |
| LD F26,-48(R1) | | ADDD F12,F10,F2 | ADDD F16,F14,F2 | | 4 |
| | | ADDD F20,F18,F2 | ADDD F24,F22,F2 | | 5 |
| SD 0(R1),F4 | SD -8(R1),F8 | ADDD F28,F26,F2 | | | 6 |
| SD -16(R1),F12 | SD -24(R1),F16 | | | | 7 |
| SD -32(R1),F20 | SD -40(R1),F24 | | | SUBI R1,R1,#48 | 8 |
| SD -0(R1),F28 | | | | BNEZ R1,LOOP | 9 |

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)

# Problems for VLIW

- Technical problems
  - Increase in code size
    - Loop unrolling—statically finding parallelism
    - Unused function slots
  - Limitations of lockstep operation
    - A stall in any function unit may cause the entire processor to stall
- Logistical problem
  - Binary code compatibility
- Major challenge for all multiple-issue processors
  - Exploit large amounts of ILP

# Advanced Compiler Support for Exploiting ILP (section 4.4 in 3rd edition) (appendix H in 6th edition)

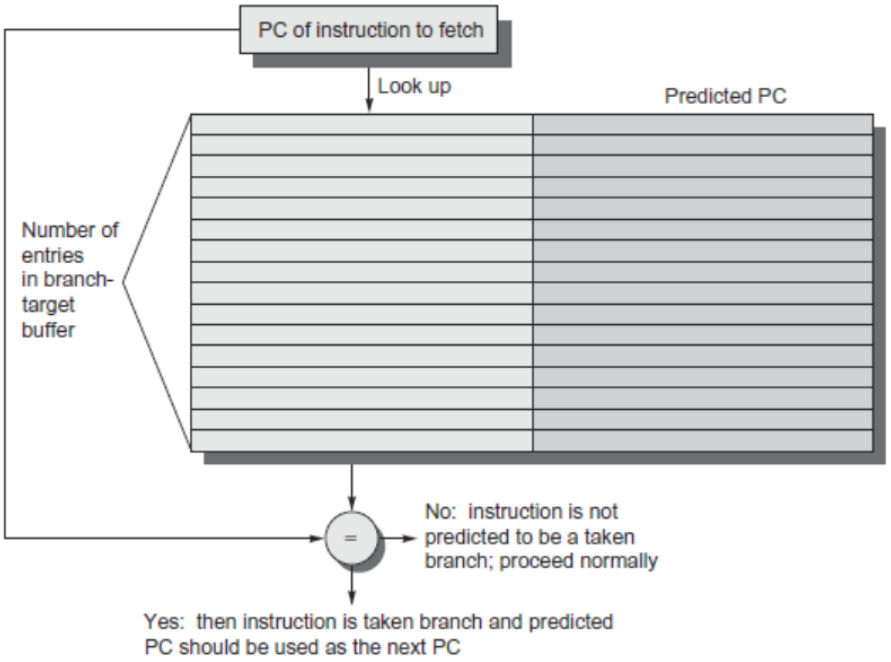
- Detecting and Enhancing Loop-level Parallelism
- Eliminating Dependent Computations
- Software pipelining: Symbolic loop unrolling
- Global Code Scheduling
  - Trace Scheduling: focus on Critical path
  - Superblocks

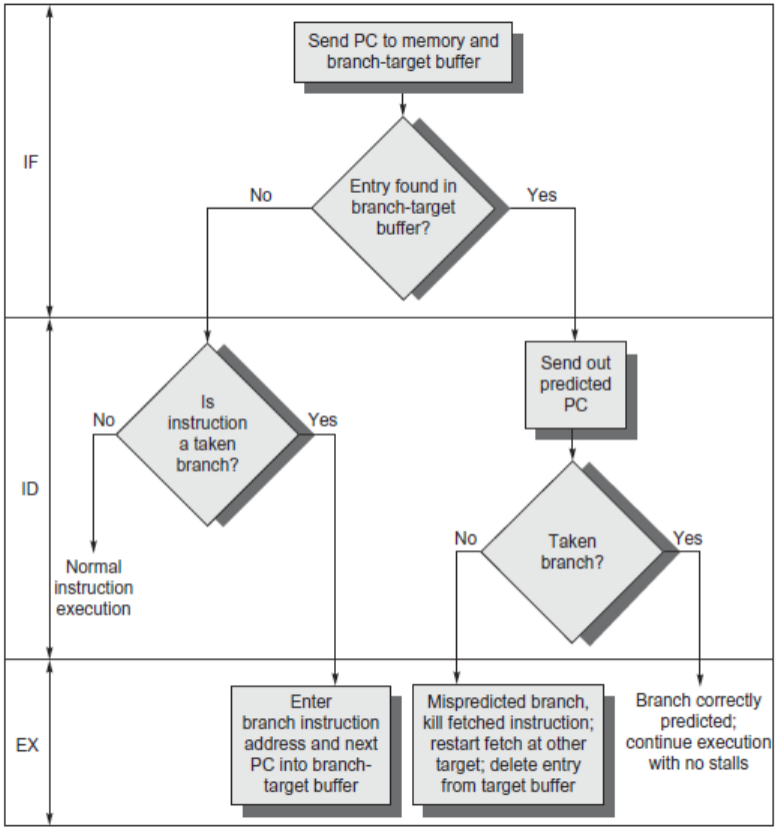
# Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
- Issue logic is the bottleneck in dynamically scheduled superscalars

# Branch-Target Buffer

- Need high instruction bandwidth
  - Branch-Target buffers
    - Next PC prediction buffer, indexed by current PC



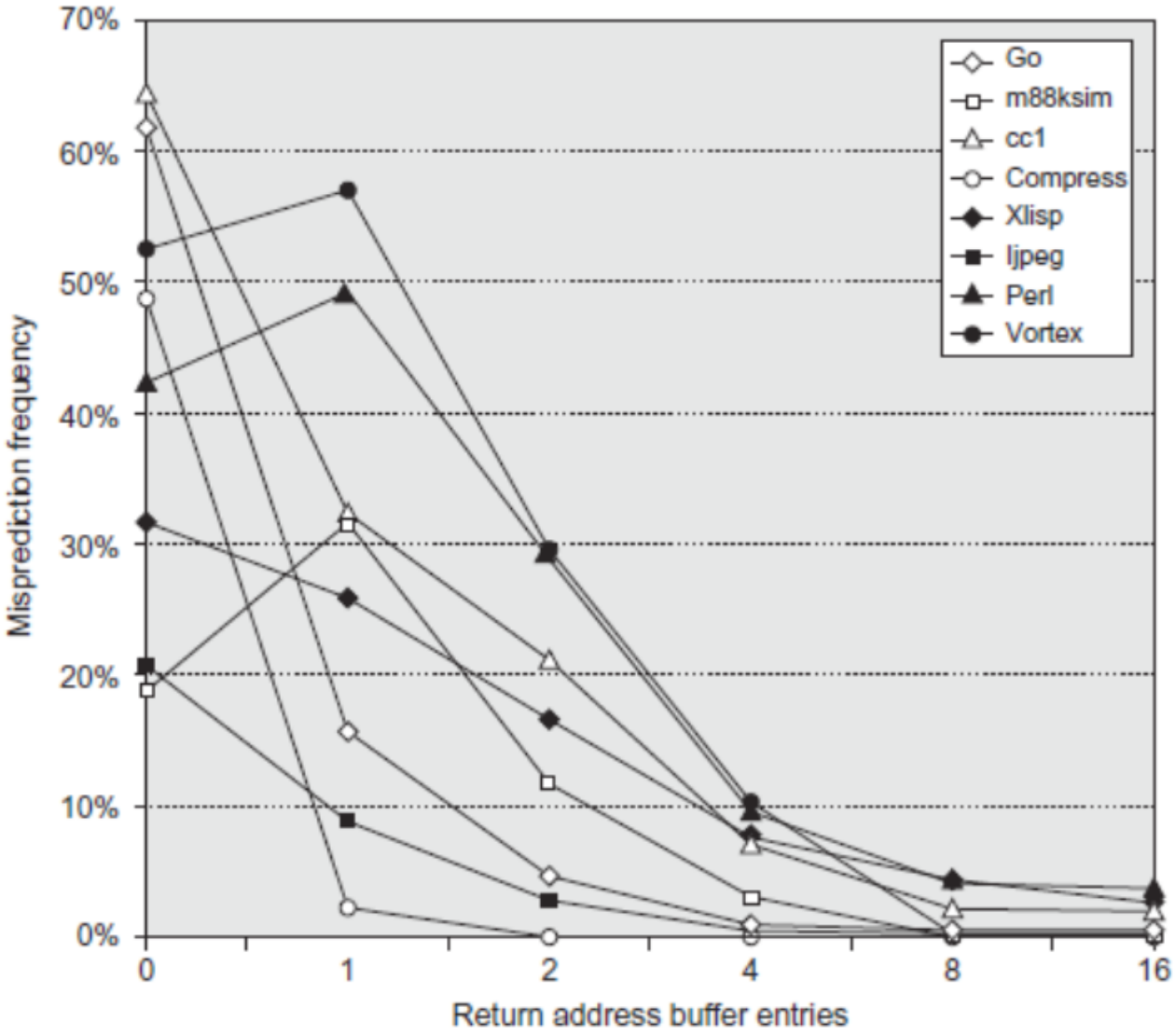
# Branch Folding

- Optimization:
  - Larger branch-target buffer
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer
  - “Branch folding”
    - combines branch instructions with predicted instructions. This technique can be implemented using an instruction queue, which buffers prefetched instructions. 、
    - PARK, Sang, Hyun, YU, Sungwook, & CHO, et al. (2005). Speculative branch folding for pipelined processors(computer systems). *ICE transactions on information and systems*.

# Return Address Predictor

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack

# Return Address Predictor



# Integrated Instruction Fetch Unit

- Design monolithic unit that performs:
  - Branch prediction
  - Instruction prefetch
    - Fetch ahead
  - Instruction memory access and buffering
    - Deal with crossing cache lines

# Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue
  - WAW and WAR hazards are avoided
  - Speculation recovery occurs by copying during commit
  - Still need a ROB-like queue to update table in order
  - Simplifies commit:
    - Record that mapping between architectural register and physical register is no longer speculative
    - Free up physical register used to hold older value
    - In other words:  SWAP physical registers on commit
  - Physical register de-allocation is more difficult
    - Simple approach:  deallocate virtual register when next instruction writes to its mapped architecturally-visibly register

# Integrated Issue and Renaming

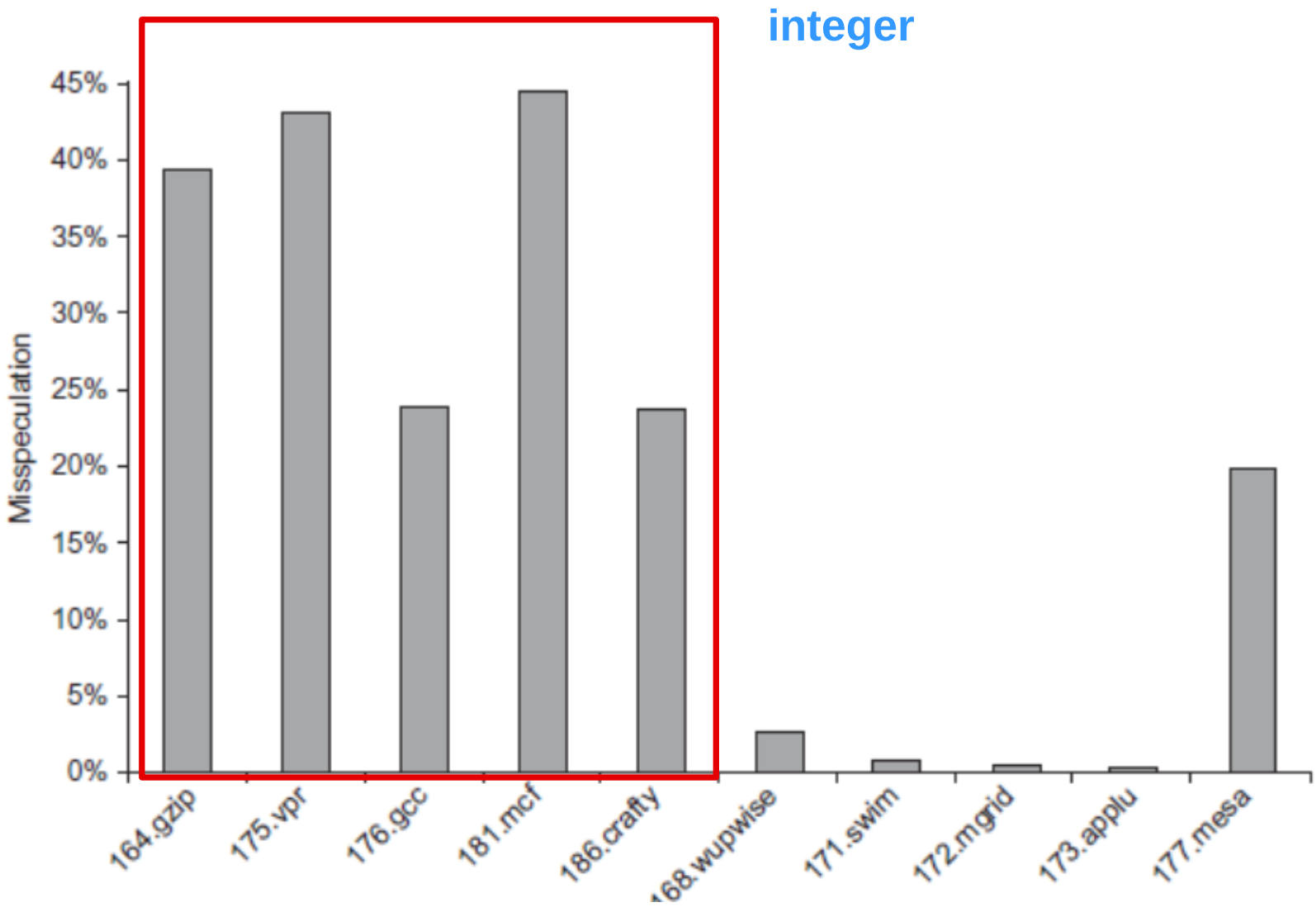
- Combining instruction issue with register renaming:
  - Issue logic pre-reserves enough physical registers for the bundle
  - Issue logic finds dependencies within bundle, maps registers as necessary
  - Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary

| Instr. # | Instruction | Physical register assigned or destination | Instruction with physical register numbers | Rename map changes |
|---|---|---|---|---|
| 1 | add x1,x2,x3 | p32 | add p32,p2,p3 | x1-> p32 |
| 2 | sub x1,x1,x2 | p33 | sub p33,p32,p2 | x1->p33 |
| 3 | add x2,x1,x2 | p34 | add p34,p33,x2 | x2->p34 |
| 4 | sub x1,x3,x2 | p35 | sub p35,p3,p34 | x1->p35 |
| 5 | add x1,x1,x2 | p36 | add p36,p35,p34 | x1->p36 |
| 6 | sub x1,x3,x1 | p37 | sub p37,p3,p36 | x1->p37 |

# How Much?

- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)
- Speculating through multiple branches
  - Complicates speculation recovery
- Speculation and energy efficiency
  - Note:  speculation is only energy efficient when it significantly improves performance

# How Much?

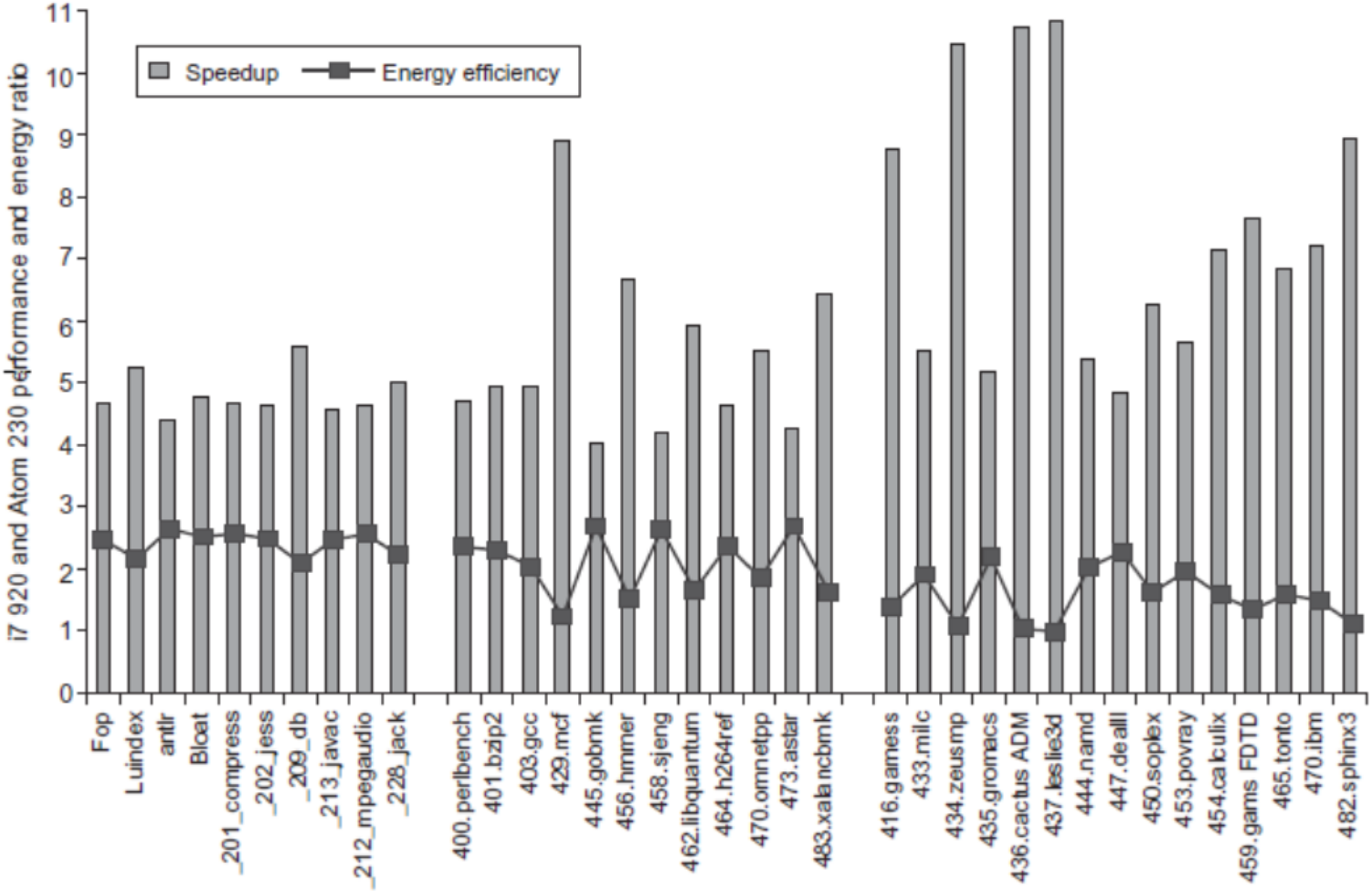

integer
Misspeculation
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
164.gzip
175.vpr
176.gcc
181.mcf
186.crafty
168.wupwise
171.swim
172.mgrid
173.applu
177.mesa

# Energy Efficiency

- Value prediction
  - Uses:
    - Loads that load from a constant pool
    - Instruction that produces a value from a small set of values
  - Not incorporated into modern processors
  - Similar idea--*address aliasing prediction*--is used on some processors to determine if two stores or a load and a store reference the same address to allow for reordering

# Fallacies and Pitfalls

- It is easy to predict the performance/energy efficiency of two different versions of the same ISA if we hold the technology constant



# Fallacies and Pitfalls

- Processors with lower CPIs / faster clock rates will also be faster

| Processor | Implementation technology | Clock rate | Power | SPECCInt2006 base | SPECCFP2006 baseline |
|---|---|---|---|---|---|
| Intel Pentium 4 670 | 90 nm | 3.8 GHz | 115 W | 11.5 | 12.2 |
| Intel Itanium 2 | 90 nm | 1.66 GHz | 104 W approx. 70 W one core | 14.5 | 17.3 |
| Intel i7 920 | 45 nm | 3.3 GHz | 130 W total approx. 80 W one core | 35.5 | 38.4 |

  - Pentium 4 had higher clock, lower CPI
  - Itanium had same CPI, lower clock

# Fallacies and Pitfalls

- Sometimes bigger and dumber is better
  - Pentium 4 and Itanium were advanced designs, but could not achieve their peak instruction throughput because of relatively small caches as compared to i7
- And sometimes smarter is better than bigger and dumber
  - TAGE branch predictor outperforms gshare with less stored predictions

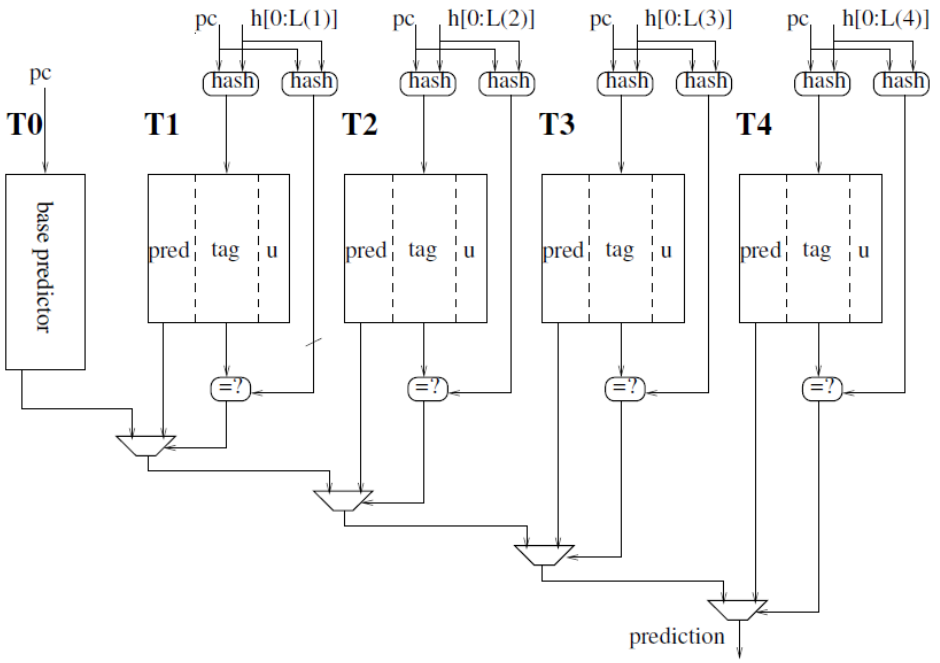
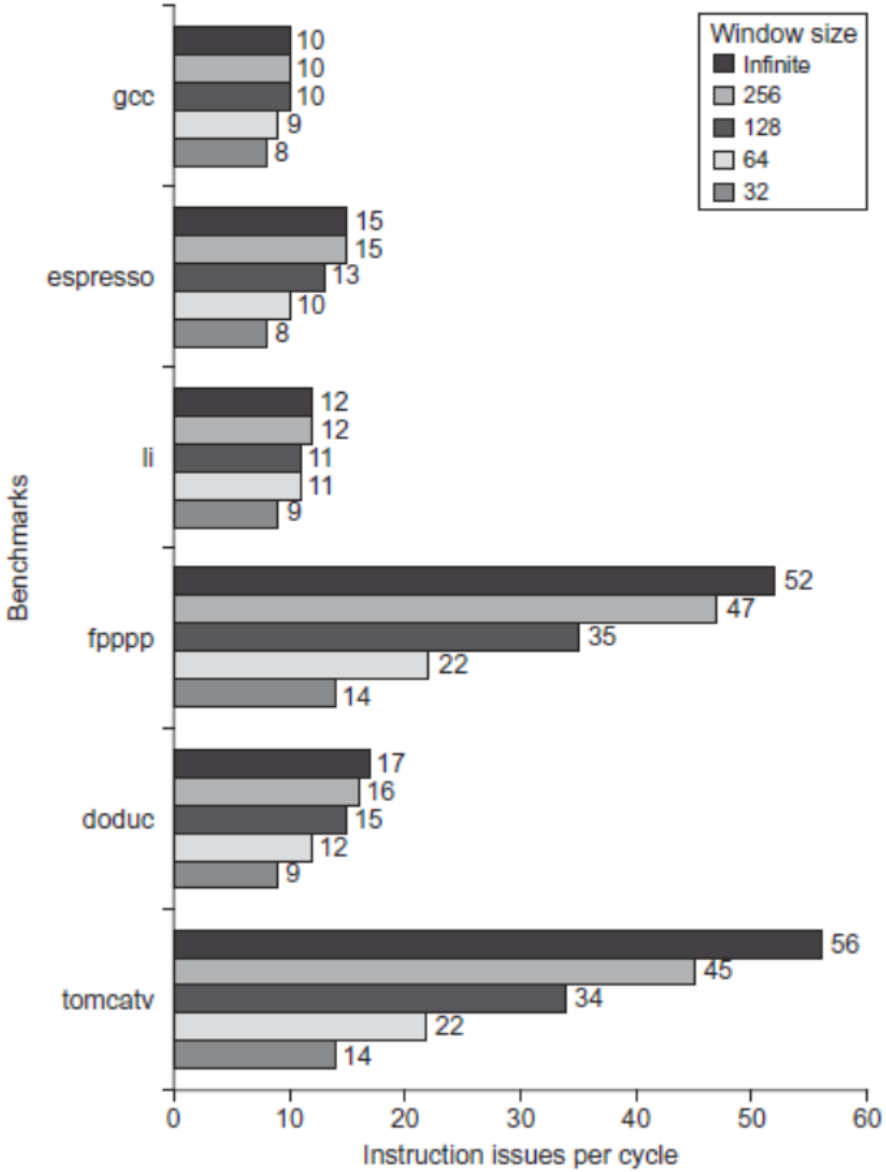


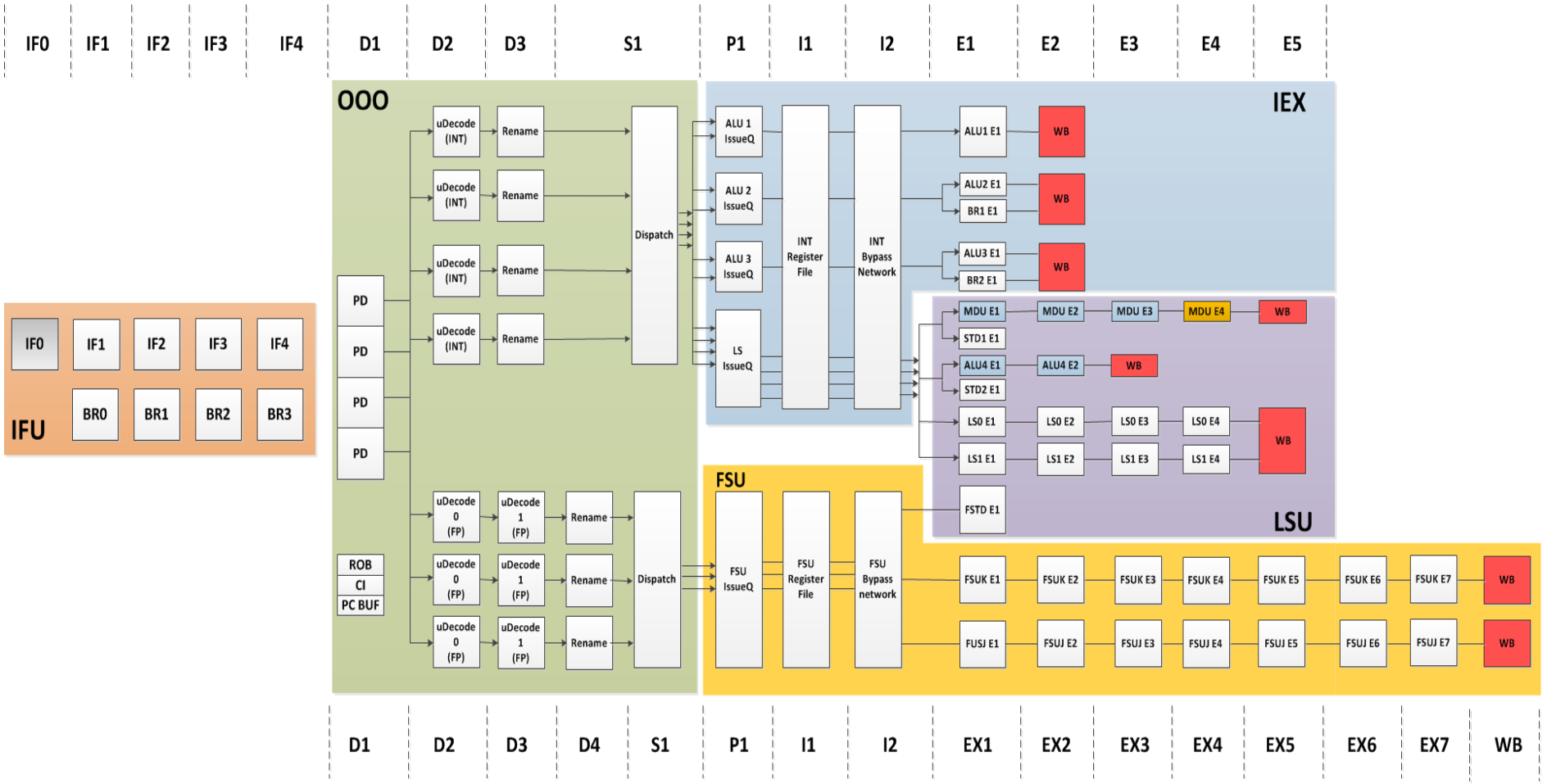

Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

# Fallacies and Pitfalls

- Believing that there are large amounts of ILP available, if only we had the right techniques



# 基于ARMv8的鲲鹏流水线技术




Taishan coreV110 Pipeline Architecture

# 基于ARMv8的鲲鹏流水线技术

- **Branch预测和取指流水线解耦设计，取指流水线每拍最多可提供32Bytes指令供译码，分支预测流水线可以不受取指流水停顿影响，超前进行预测处理；**
- **定浮点流水线分开设计，解除定浮点相互反压，每拍可为后端执行部件提供4条整型微指令及3条浮点微指令；**
- **整型运算单元支持每拍4条ALU运算（含2条跳转）及1条乘除运算；**
- **浮点及SIMD运算单元支持每拍2条ARM Neon 128bits 浮点及SIMD运算；**
- **访存单元支持每拍2条读或写访存操作，读操作最快4拍完成，每拍访存带宽为2x128bits读及1x128bits写；**

# Dynamic Scheduling in P6 (Pentium Pro, II, III)

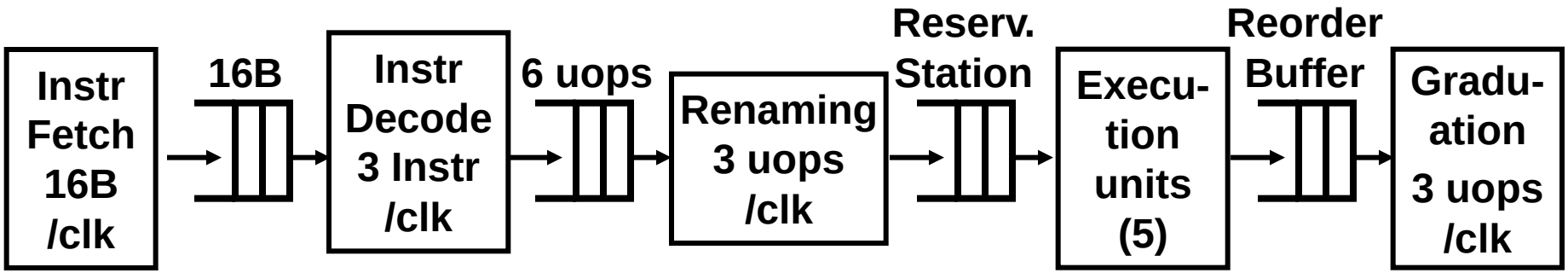
- Q: How pipeline 1 to 17 byte 80x86 instructions?
- P6 doesn't pipeline 80x86 instructions
- P6 decode unit translates the Intel instructions into 72-bit micro-operations (~ MIPS)
- Sends micro-operations to reorder buffer & reservation stations
- Many instructions translate to 1 to 4 micro-operations
- Complex 80x86 instructions are executed by a conventional microprogram (8K x 72 bits) that issues long sequences of micro-operations
- 14 clocks in total pipeline (~ 3 state machines)

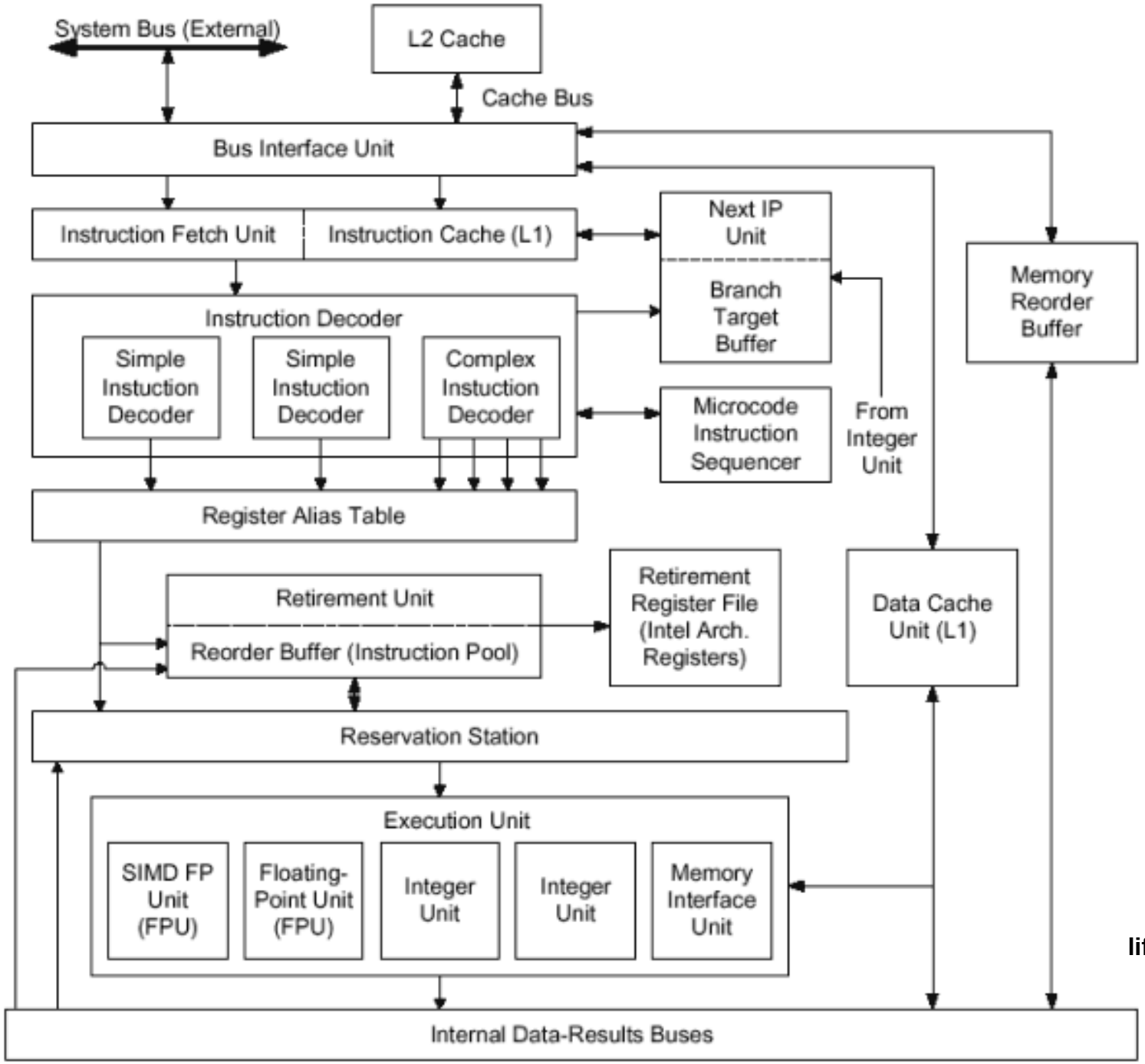
# Dynamic Scheduling in P6

| Parameter | 80x86 | micro ops |
|---|---|---|
| Max. instructions issued/clock | 3 | 6 |
| Max. instr. complete exec./clock | | 5 |
| Max. instr. commited/clock | | 3 |
| Window (Instrs in reorder buffer) | | 40 |
| Number of reservations stations | | 20 |
| Number of rename registers | | 40 |
| No. integer functional units (FUs) | | 2 |
| No. floating point FUs | | 1 |
| No. SIMD Fl. Pt. FUs | | 1 |
| No. memory Fus | | 1 load + 1 store |

# P6 Pipeline

- 14 clocks in total (~3 state machines)
- 8 stages are used for in-order instruction fetch, decode, and issue
  - Takes 1 clock cycle to determine length of 80x86 instructions + 2 more to create the micro-operations (uops)
- 3 stages are used for out-of-order execution in one of 5 separate functional units
- 3 stages are used for instruction commit



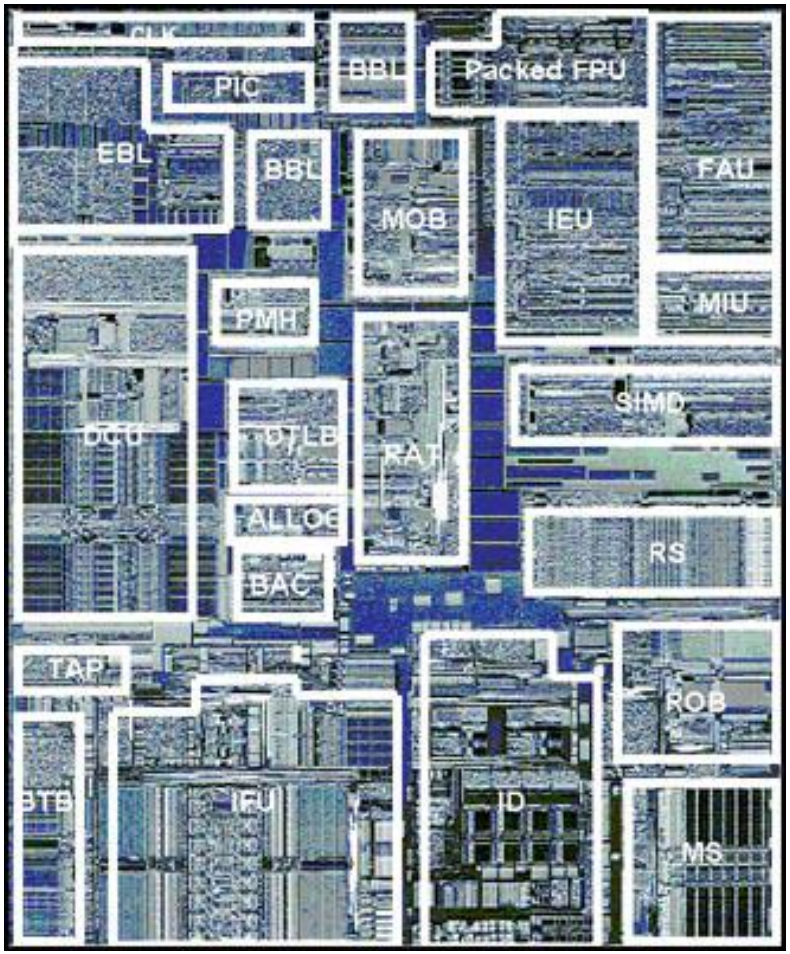



ram

- IP = PC

**From: http://www.digit-life.com/articles/pentium4/**

# Pentium III Die Photo




**1st Pentium III, Katmai: 9.5 M transistors, 12.3 * 10.4 mm in 0.25-mi. with 5 layers of aluminum**

- EBL/BBL - Bus logic, Front, Back
- MOB - Memory Order Buffer
- Packed FPU - MMX Fl. Pt. (SSE)
- IEU - Integer Execution Unit
- FAU - Fl. Pt. Arithmetic Unit
- MIU - Memory Interface Unit

---

- DCU - Data Cache Unit
- PMH - Page Miss Handler
- DTLB - Data TLB
- BAC - Branch Address Calculator
- RAT - Register Alias Table
- SIMD - Packed Fl. Pt.
- RS - Reservation Station

---

- BTB - Branch Target Buffer
- IFU - Instruction Fetch Unit (+I$)
- ID - Instruction Decode
- ROB - Reorder Buffer
- MS - Micro-instruction Sequencer

# END