Current Microprocessors

Pipeline

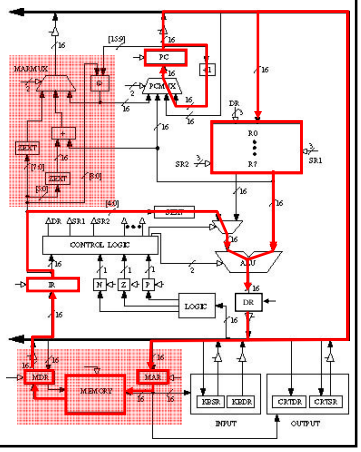
Efficient Utilization of Hardware Blocks

- Execution steps for an instruction:
  1. Send instruction address (IA)
  2. Instruction Fetch (IF)
  3. Store instruction (SI)
  4. Decode Instruction, fetch operands (DI)
  5. Address Computation (AC)
  6. Memory Access (ME)
  7. Execution (EX)
  8. Write Back (WB)

0001	101	100	1	00011

ADD R5, R4, #3

Only one block used every cycle

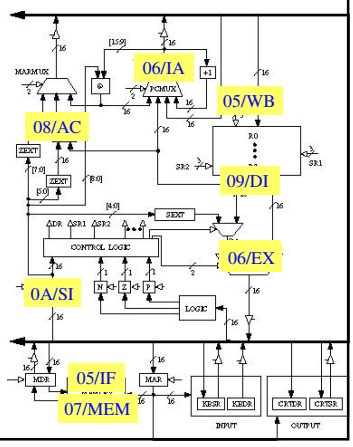


Efficient Utilization of Hardware Blocks

```
for (i=0; i < 100; i++) {
   a[i] = a[i] + 5
}
```

```
            ...
05 LOOP     LDR R1, R0, #3
06          ADD R1, R1, #5
07          STR R1, R0, #30
08          ADD R0, R0, #1
09          ADD R3, R0, R2
0A          BRn LOOP
            ...
```

- **All blocks used**
- **One instruction terminates each cycle**



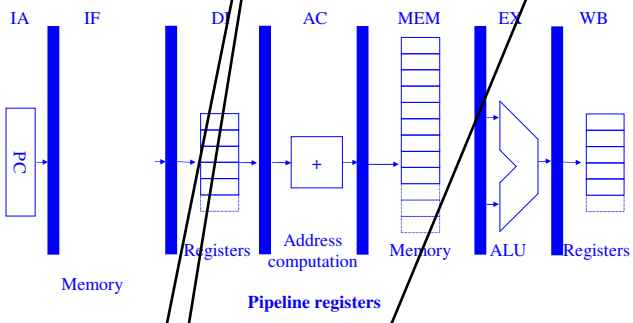
Pipeline

All blocks used

Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12	13
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB						
ADD R1,R1,#5		IA	IF	SI	DI	AC	MEM	EX	WB					
STR R1,R0,#30			IA	IF	SI	DI	AC	MEM	EX	WB				
ADD R0,R0,#1				IA	IF	SI	DI	AC	MEM	EX	WB			
ADD R3,R0,R2					IA	IF	SI	DI	AC	MEM	EX	WB		
BNE LOOP						IA	IF	SI	DI	AC	MEM	EX	WB	
LDR R1,R0,#30							IA	IF	SI	DI	AC	MEM	EX	WB
ADD R2,R1,#5								IA	SI	IF	DI	AC	MEM	EX

- Program execution up to **8 times faster**.
- Instruction execution time barely changed.

Pipeline Implementation



- All information (data and control) stored in pipeline registers

Trends

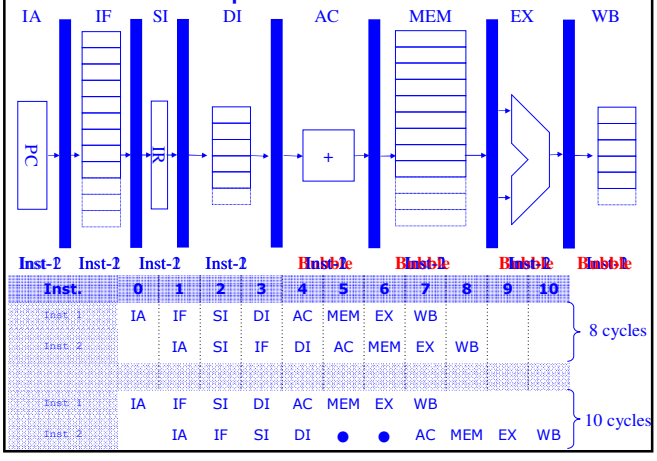
- The Pentium IV pipeline has 20 stages + 8 conversion stages `x86 → µInstructions`.

- The initial motivation for the pipeline was:
  - Efficiently exploiting architecture blocks
  - Increase instruction execution rate
- Current motivation is increasing clock frequency:
  - Split stages into sub-stages
  - Maximum duration of a sub-stage reduced
    Enables clock frequency increase
- Difficult to reach **sustained** performance

Pipeline Hazards

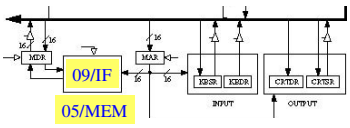
- Not always possible to issue/commit one instruction per cycle
- When an instruction cannot proceed it is a pipeline **hazard**
- Three types of **pipeline hazards**:
  - Resource hazard
  - Data hazard
  - Control hazard
- Hazard induces a pipeline **stall**
- The control circuit injects one or several **bubbles**
- Performance metric: IPC (*Instructions Per Cycle*)
  < 1 (optimum) if hazards

Pipeline Hazard



Resource Hazards

- Access same block at same cycle
- Solutions:
  - Accept pipeline stall (SPARC)
  - Increase resources (memory ports)



Inst.	0	1	2	3	4	5
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM
ADD R1,R1,#5		IA	IF	SI	DI	AC
STR R1,R0,#30			IA	IF	SI	DI
ADD R0,R0,#1				IA	IF	SI
ADD R3,R0,R2					IA	IF

Resource conflict

Resource Hazards

No memory access — no resource hazard

Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB					
ADD R1,R1,#5		IA	IF	SI	DI	AC	MEM	EX	WB				
STR R1,R0,#30			IA	IF	SI	DI	AC	MEM	EX	WB			
ADD R0,R0,#1				IA	IF	SI	DI	AC	MEM	EX	WB		
ADD R3,R0,R2					IA	●	IF	SI	DI	AC	MEM	EX	WB
BRn LOOP						●	IA	●	IF	SI	DI	AC	MEM
LDR R1,R0,#30								●	IA	IF	SI	DI	AC
ADD R1,R1,#5										IA	IF	SI	DI

Data Hazards

- Data dependence between two instructions

```
LDR R1 ← R0,#30
ADD R1 ←→ R1,#5
```

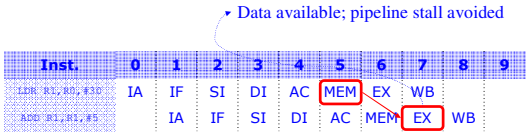
- An instruction can fetch operands only when available

R_1 contains value expected by LDR

Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB					
ADD R1,R1,#5		IA	IF	SI	●	●	●	●	DI	AC	MEM	EX	WB

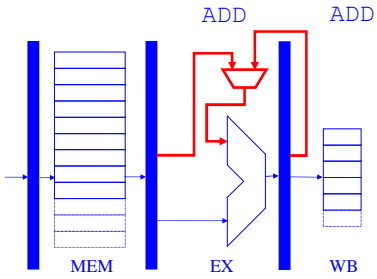
Forwarding

- Data is often available in processor before it is written in register:
  Immediately pass data to expecting block
  = ***Forwarding***
- Pipeline stall only when data absolutely necessary



Implementing *forwarding*

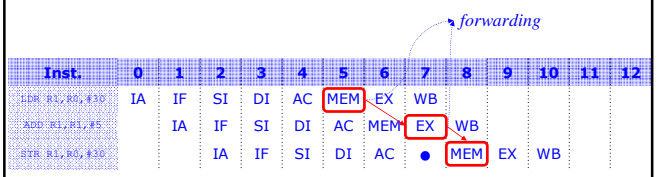
- *Forwarding* requires:
  - Additional data paths
  - Adding/Increasing size of muxes
  - Modifying control circuit (detect/activate *forwarding*)



Forwarding

- *Forwarding* cannot avoid all pipeline stalls:

```
ADD R1 ← R1,#5
STR R1 → R0,#30
```



Multi-Cycles Instructions

- Example: floating-point instructions
  - FPADD: 2 execution cycles
  - FPMUL: 5 execution cycles

Resource hazard if FPADD and FPMUL in same block

Inst.	0	1	2	3	4	5	6	7				
FPMUL F2,F1,F0	IA	IF	SI	DI	AC	MEM	EX	**EX**	**EX**	**EX**	**EX**	WB
FPADD F4,F2,F3		IA	IF	SI	DI	AC	MEM	●	●	●	●	EX	EX
FPADD F2,F1,F3			IA	IF	SI	DI	AC	MEM	EX	**EX**	WB	

Ecriture dans le désordre

- New data dependences:
  - Write registers out of order
- New resource conflicts (register banks ports)

Inst.	0	1	2	3	4	5	6	7					
FPMUL F2,F1,F0	IA	IF	SI	DI	AC	MEM	EX	EX	EX	EX	EX	WB	
FPADD F4,F2,F3		IA	IF	SI	DI	AC	MEM	●	●	●	●	EX	EX
FPADD F5,F1,F3			IA	IF	IF	DI	AC	MEM	EX	EX	●	●	WB

Pipeline and Exceptions

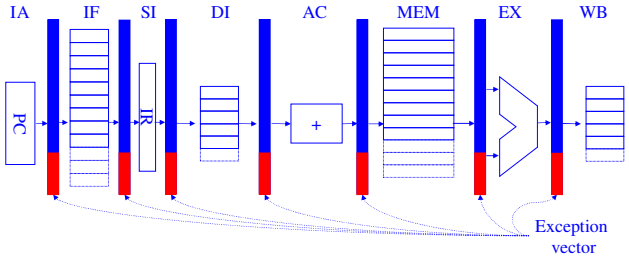
- Le pipeline makes exception management harder. Example:
  - `LDR` has a page fault in `MEM`
  - `ADD` has a page fault in `IF`
- Precise exception on instruction I:
  - All instructions before I finish normally
  - All instructions after I can be interrupted, then reexecuted from the start after exception handled

Exceptions

Inst.	0	1	2	3	4	5	6	7	8
LDR	IA	IF	SI	DI	AC	MEM	EX	WB	
ADD		IA	SI	IF	DI	AC	MEM	EX	WB

- Necessary to implement precise exceptions

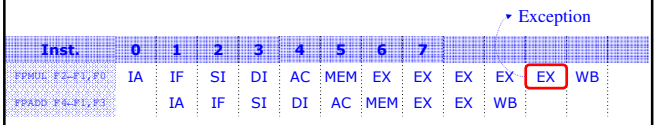
Pipeline and Exceptions



- **Exception vector** for each pipeline register
- After exception, no more state modification
- In WB, exceptions dealt with in same order as instructions

Multi-Cycle Instructions and Exceptions

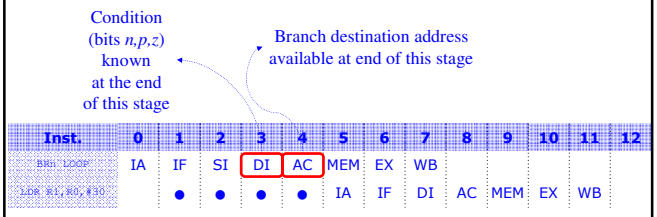
- Example:
  - `FPADD` does exception `NaN` in `EX`
- Processor state modified in `FPADD` before exception detection
- Forbid **out of order** state modification



Control Hazards

- Branch: must know destination (and possibly condition value) before fetching next instruction

```
LOOP    LDR R1 ← R0,#30
        ...
        BRn  LOOP
```

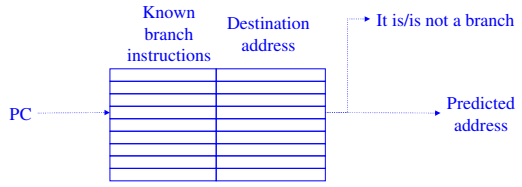


Current Microprocessors

Branch Prediction

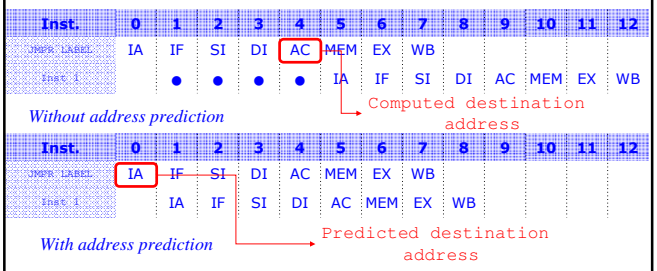
Branch Prediction

- Usually, branch destination address is constant (except for `RET` and indirect branches)
- **Predict** destination address:
  - store destination addresses in a table for each branch execution
  - table is indexed by branch instruction PC
  - when PC sent to memory, also sent to table
  - table says if it's a branch, and the destination address



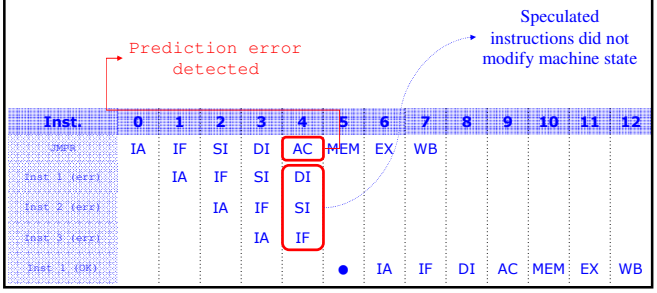
Address Prediction

- If PC corresponds to branch, update PC: `PC = Destination address`
- Example: conditional branch

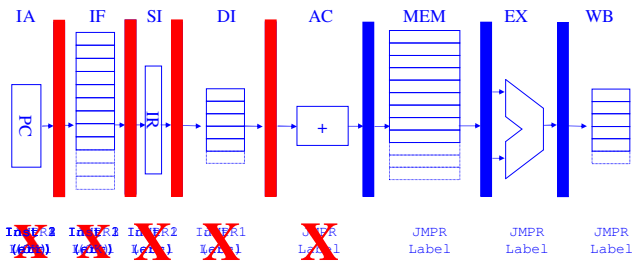


Error Prediction

- Detect error (destination address always computed)
- Squash speculatively fetched instructions
- Speculated instructions only modify machine state after check
- Branch squashing costs 1 cycle



Recovery After Misprediction



Condition Prediction

- Conditional branches
- Must predict condition value
- Condition value can change often from one branch execution to another **prediction difficult**
- Example: branch taken

Inst.	0	1	2	3	4	5	6	7	8	9
BR LABEL	IA	IF	SI	DI	AC	MEM	EX	WB		
Inst 1		●	●	●	IA	IF	DI	AC	MEM	EX	WB

Condition computed

No condition prediction, address prediction

Inst.	0	1	2	3	4	5	6	7	8	9
BR LABEL	IA	IF	SI	DI	AC	MEM	EX	WB		
Inst 1		IA	IF	SI	DI	AC	MEM	EX	WB	

Condition predicted

With condition and address prediction

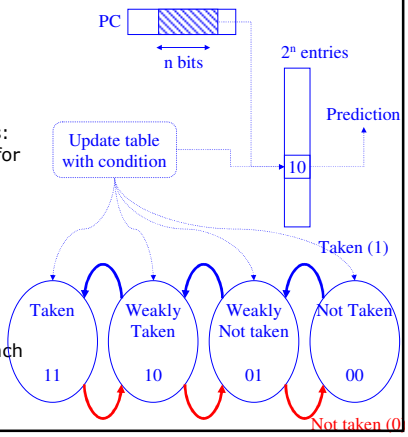
Prediction Strategies

- Static prediction:
  - Always taken
    - Works well with loops
  - Compile-Time analysis
    - EPIC/IA-64; limitations of static analysis
  - Hit rate: ≈ from 70% to 90%

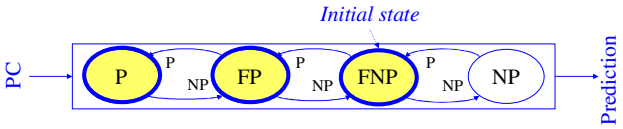
```
                ...
05  LOOP        LDR R1, R0, #3
06              ADD R1, R1, #5
07              STR R1, R0, #30
08              ADD R0, R0, #1
09              ADD R3, R0, R2
0A              BRn LOOP
                ...
```

Prediction Strategies

- **Dynamique** prediction:
  - Commonplace in processors
  - Recent mechanisms: hit rate up to 99% for certain applications
  - Principle: learn individual branch behavior
- A first mechanism: local history
  - one 4-state automaton per branch



Example



i=0 (taken)	not taken
i=1 (taken)	taken
i=2 (taken)	taken
i=3 (taken)	taken
...	...
i=99 (non taken)	taken

```
...
05 LOOP   LDR R1, R0, #3
06        ADD R1, R1, #5
07        STR R1, R0, #30
08        ADD R0, R0, #1
09        ADD R3, R0, R2
0A        BRn LOOP
...
```

100 iterations:

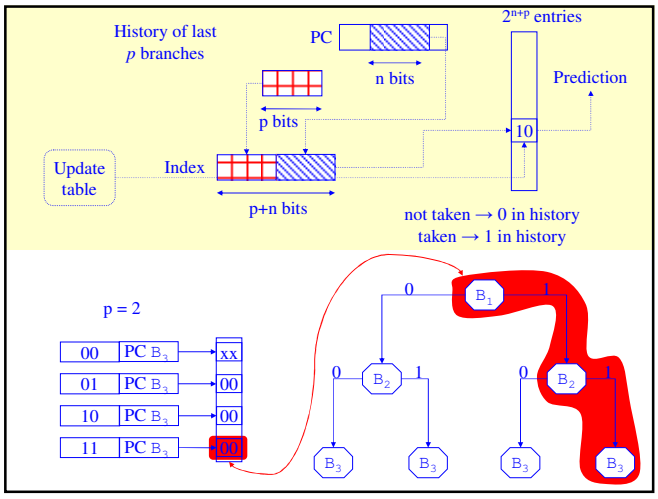
- iteration i=0: BRn taken
- iteration i=1: BRn taken
- ...
- iteration i=99: BRn not taken

Improving Dynamic Prediction

- A small hit rate increase can have a significant impact on overall processor performance

```
if (a == 1) a = 0;      /* Branchement B1 */
...
if (b == 0) b = 1;      /* Branchement B2 */
...
if (a == b) ...;        /* Branchement B3 */
```

- To improve prediction accuracy, use behavior of preceding branches: **global history**



Impact of Branch Prediction on Processor Performance

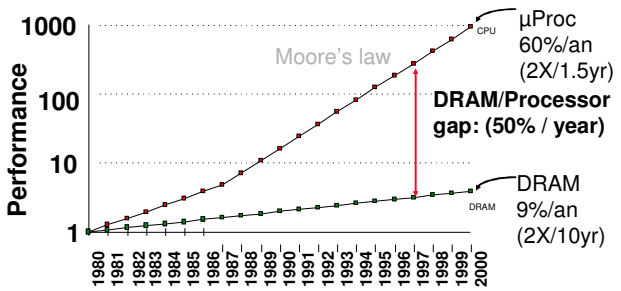
- On average, 1 instruction out of 5 is a branch
- In current pipelines, misprediction cost is 5 to 10 cycles

- 8 cycles between fetch and branch resolution
- 4 cycles to restart the pipeline
  12 cycles penalty
- 1 instruction / cycle (1000 instructions) :
  - 50% wrong predictions:
    1000 * (0,8*1 + 0,2*(0,5*1 + 0,5*12)) = 2100 cycles
  - 20% wrong predictions:
    1000 * (0,8*1 + 0,2*(0,8*1 + 0,2*12)) = 1440 cycles
  - 5% wrong predictions:
    1000 * (0,8*1 + 0,2*(0,95*1 + 0,05*12)) = 1110 cycles
- 5 instructions / cycle (1000 instructions) :
  - 50% wrong predictions:
    200 * (0,5*1 + 0,5*12) = 1300 cycles
  - 20% wrong predictions:
    200 * (0,8*1 + 0,2*12) = 640 cycles
  - 5% wrong predictions:
    200 * (0,95*1 + 0,05*12) = 310 cycles

Current Microprocessors

Caches

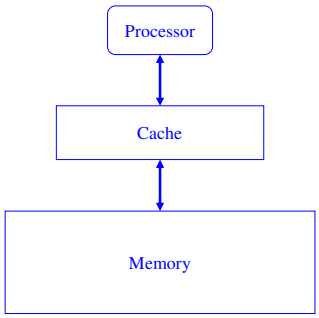
Memory Access Latency



- Processor cycle time << memory access time

Cache Memory

- Fast (SRAM) but small (cost) memory:
  - ≈ 1 to 3 cycles (pipelined)
- Processor sends memory requests to cache
  - Data in cache: hit
  - Data not in cache: miss
- Performance:
  - Hit rate
  - Average memory access time



1-bit SRAM Cell

- SRAM=*Static Random Access Memory.*
- Writing:
  - bit=*va*

Caches and Locality Properties

- Most programs have strong locality properties
- **Temporal** locality: address *A* referenced at time *t* has strong probability to be referenced again within a short time interval
- **Spatial** locality: address *A* referenced at time *t*, strong probability to reference a neighbor address within a short time interval

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    y[i] = y[i] + a[i][j] * x[j]
  }
}
```

- `y[i]`: spatial and temporal locality
- `a[i][j]`: spatial locality
- `x[j]`: spatial and temporal locality

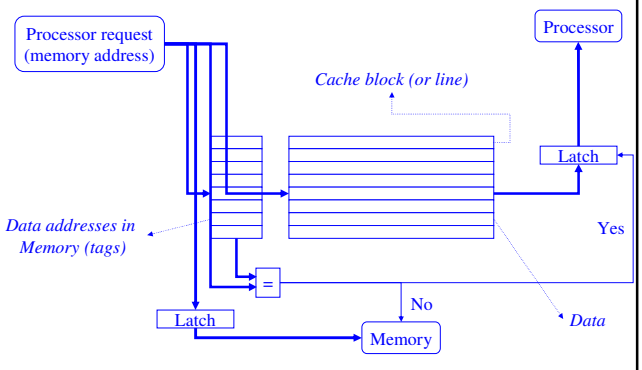
Data and Instructions Locality

- Locality properties for instructions as well
- Temporal locality: just keep address in cache
- Sptial locality: load addresses by blocks

```
                ...
05  LOOP        LDR R1, R0, #3
06              ADD R1, R1, #5
07              STR R1, R0, #30
08              ADD R0, R0, #1
09              ADD R3, R0, R2
0A              BRn LOOP
                ...
```

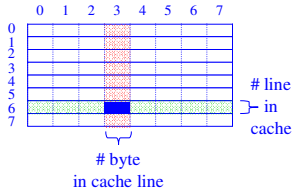
Loop
reuse instructions
temporal locality
Consecutive instructions
spatial locality

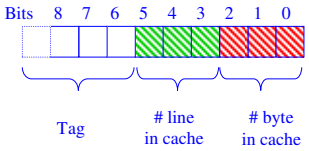
Cache Architecture



Address Mapping

- Hardware mapping
  - Programmer does not manage it
  - Transparent for programmer
- Mapping is a simple function of the address
  - # line in cache
  - # byte in line

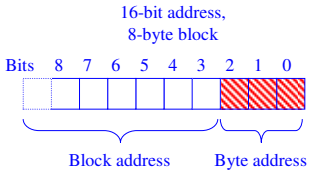




- C_S bytes cache
- L_S bytes cache
- # byte: $\log_2(L_S)$ least significant bits of address
- # line: $\log_2(C_S/L_S)$ least significant bits of address

Cache Line

- Fetch data by blocks
- Address = byte address
- Block:
  - Most significant bits of bytes addresses identical
  - Only least significant bits vary

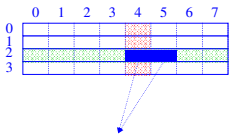


Example:
0...0101000**10**
0...0101001**11**
0...0101011**11**

Same cache line
Distinct lines
Consecutive addresses

Reading Data

- Example:
  - C_S = 32 bytes
  - L_S = 8 bytes
- Requested address (16 bits):
  - 0110010001010**100**
  - # line: **10**
  - # byte in line: **100**
- A request can have a variable size:
  - byte, half-word, word...
  - request = address + nb of bytes
  - address = addresse of first byte
  - example: 2 bytes (16-bit word)

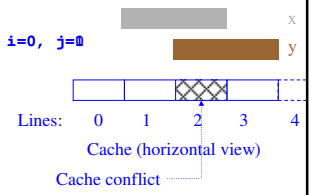


Associativity

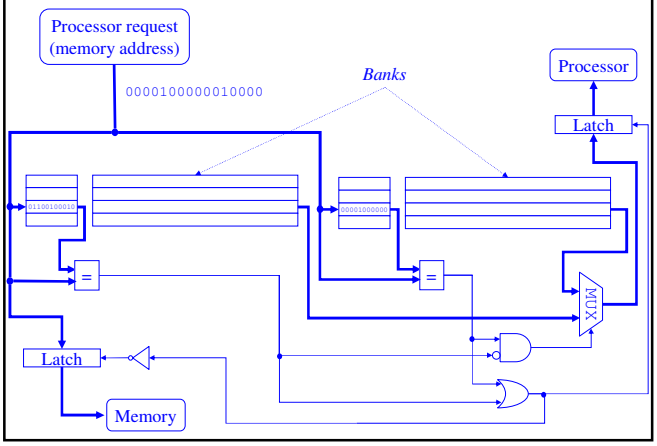
- Physical memory size >> cache size
- Mapping function can breed data conflicts
- Reduce conflicts by increasing cache **associativity**

```
for (i=0; i<N; i++) {
  for (j=0; j<2; j++) {
    x[j] = y[j]
  }
}
```

```
@x = 0110010001001000
@y = 0000100000010000
```

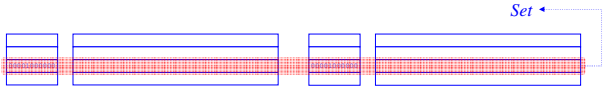


Associative Cache Structure

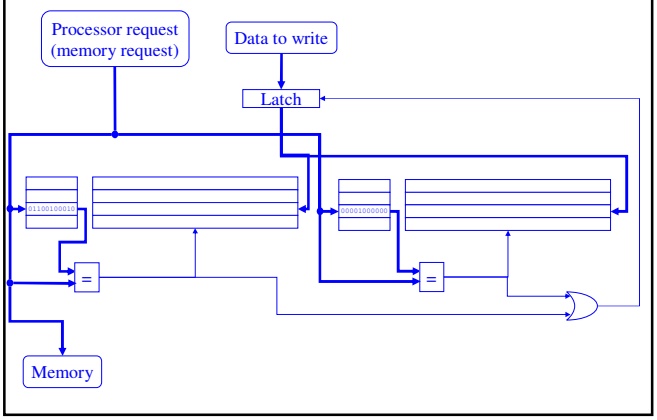


Associative Cache Operations

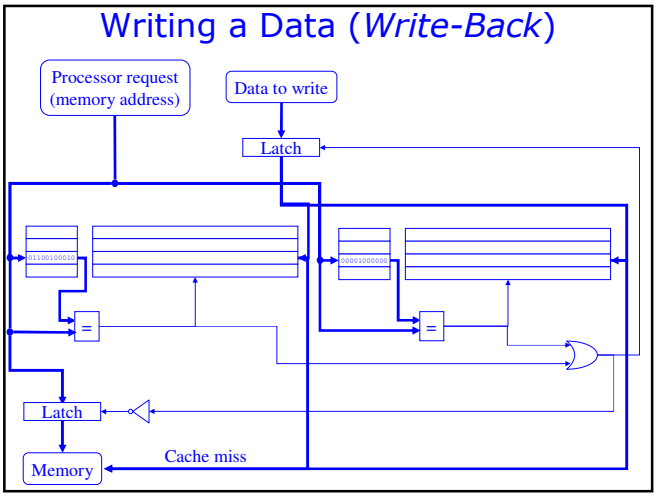
- Degree of associativity *n*.
- A data can be stored within *n* different entries
- Upon a cache miss, choose block/bank
- Set of possible blocks = **set**:
  - LRU: Least Recently Used
  - Random
  - Pseudo-LRU: most recently used line not replaced, random among others
  - FIFO: First In First Out



Writing a Data (*Write-Through*)

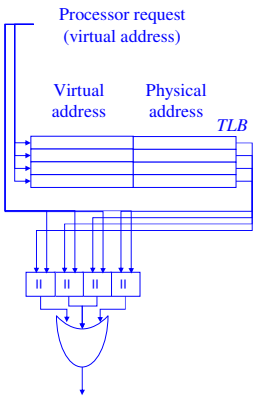


Writing a Data (*Write-Back*)

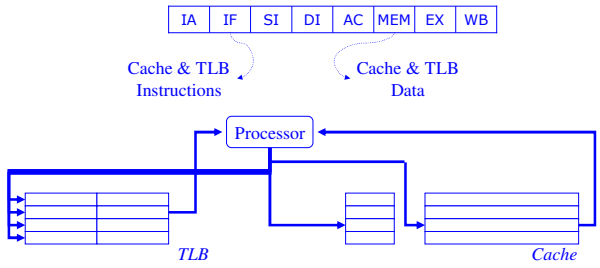


Virtual Memory/Physical Memory: TLB

- Processor uses *virtual* addresses
- Data have an address in *physical* memory
  - Virtual/Physical address translation
  - **TLB** (*Translation Lookaside Buffer*)
- Cache of address translations
- 1 TLB entry = 1 page
- TLB often fully associative (*n* = number of lines).



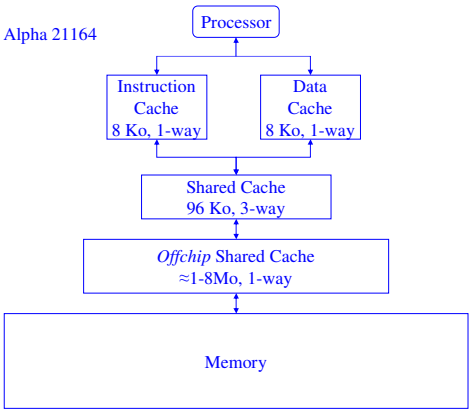
Summary



- Simultaneous:
  - Virtual/physical address translation
  - Cache access using virtual address

Several Cache Levels

Example: Alpha 21164



Impact of Cache Misses on Processor Performance

- On average, 1/3 instructions are load/store
- Instruction cache misses
- Cache hierarchies reduce average memory latency

•1GHz processor, 100ns for memory access
• 1000 instructions :
  • 50% cache misses:
    1000 * (0,67*1 + 0,33*(0,5*1 + 0,5*100)) = 17335 cycles
  • 5% cache misses:
    1000 * (0,67*1 + 0,33*(0,95*1 + 0,05*100)) = 2633 cycles
  • 0% cache misses:
    1000 cycles

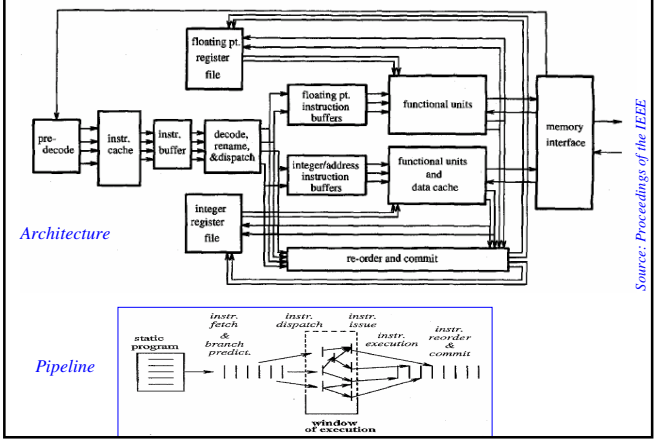
Current Microprocessors

Superscalar Execution

Superscalar Processor

- Pipeline: at most one instruction per cycle
- Superscalar degree of *n*: up to *n* instructions complete per cycle (in practice, $n \approx 4$)
- Requirements for a superscalar processor:
  - An uninterrupted flow of instructions
  - Determine which instructions can execute in parallel
  - Propagate data among instructions (result of instruction *i* is operand of instruction *j*)
  - Several functional units
- Constraint: precise interruptions
- Superscalar implemenations share a lot of features

Superscalar Processor Architecture



Architecture

Source: Proceedings of the IEEE

Pipeline

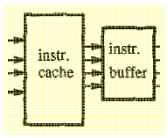
Instruction-Level Parallelism (*ILP*)
(fine-grain parallelism)

```
for (i=0; i<last; i++) {
  if (a[i] > a[i+1]) {
    temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    change++;
  }
}
```

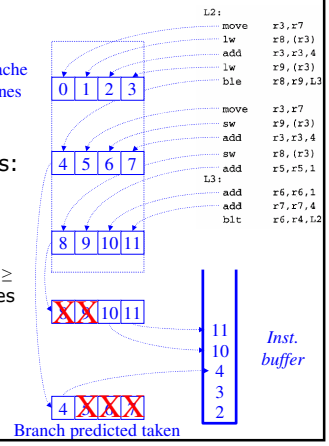
```
L2:
   move    r3,r7      #r3->a[i]
   lw      r8,(r3)    #load a[i]
   add     r3,r3,4    #r3->a[i+1]
   lw      r9,(r3)    #load a[i+1]
   ble     r8,r9,L3   #branch a[i]>a[i+1]

   move    r3,r7      #r3->a[i]
   sw      r9,(r3)    #store a[i]
   add     r3,r3,4    #r3->a[i+1]
   sw      r8,(r3)    #store a[i+1]
   add     r5,r5,1    #change++
L3:
   add     r6,r6,1    #i++
   add     r7,r7,4    #r4->a[i]
   blt     r6,r4,L2   #branch i<last
```

Instruction Fetch

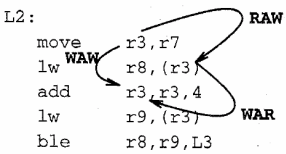


- Instruction flow disruptions:
  - branches
  - instruction cache misses
- Avoid disruptions:
  - Number fetched instructions ≥ 4: possibly several cache lines (multi-port cache)
  - Buffer to store pre-fetched instructions



Dependences

- Find instruction dependences
- Avoid "false" dependences due to register aliasing



- RAW (*Read After Write*): true dependence
- WAW (*Write After Write*): out-of-order write (false dependence)
- WAR (*Write After Read*): too early write (false dependence)

Register Aliasing: Renaming

- Binary compatibility limits number of registers
- Technology allows more registers
- Physical registers + ReOrder Buffer (ROB)
- Each instruction mapped to a ROB entry
- A table maps logical registers to: either a physical register or a ROB entry

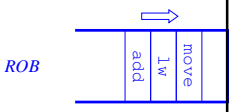
  Eliminates register aliasing

  Finds true dependences

```
L2:
    move    r3, r7
    lw      r8, (r3)
    add     r3, r3, 4
```

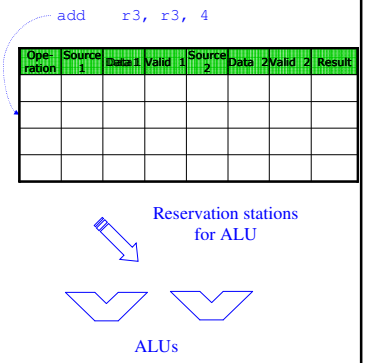
Logical register	Physical storage
r_3	ROB_6
...	...
r_8	ROB_7

Physical register	Value
r_3	produced by move
...	...
r_8	produced by lw



Dispatch

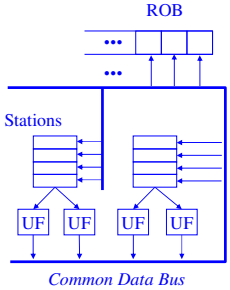
- After dependences, dispatch
- Reservation stations: buffer for each function unit
- Instruction executed when:
  - all operands available
  - functional unit available
- Tomasulo algorithm



Issue

- Ready instructions sent to FU
- Result propagated through buses:
  - to physical storage (ROB entries)
  - to reservation stations
- Pending instructions immediately issued

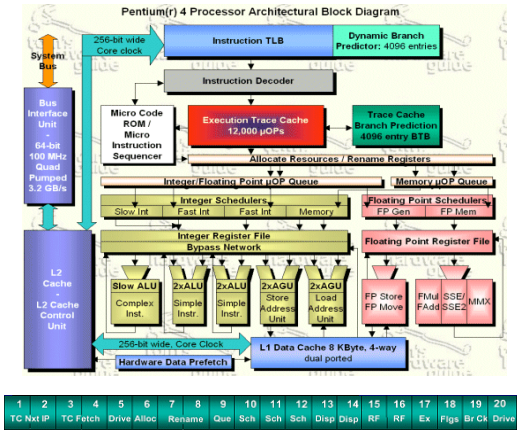
  Internal model closer to *dataflow* than *von Neumann*



Commit

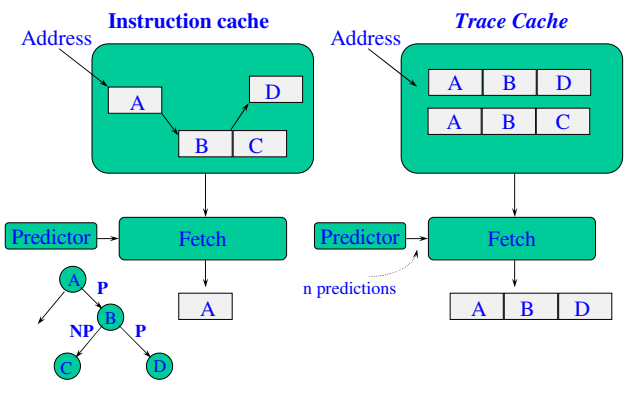
- An instruction can only commit when reaching ROB end
  commit order = program order
- Logical architecture state only modified at commit
  precise interruptions possible in OoO processors
- Logical state: registers and memory
- When an instruction leaves the ROB:
  - result written into register
  - OR data sent to memory
- Superscalar processor of degree *n*: *n* instructions can commit simultaneously
- If instruction at top of ROB has not completed, processor is stall

Pentium IV

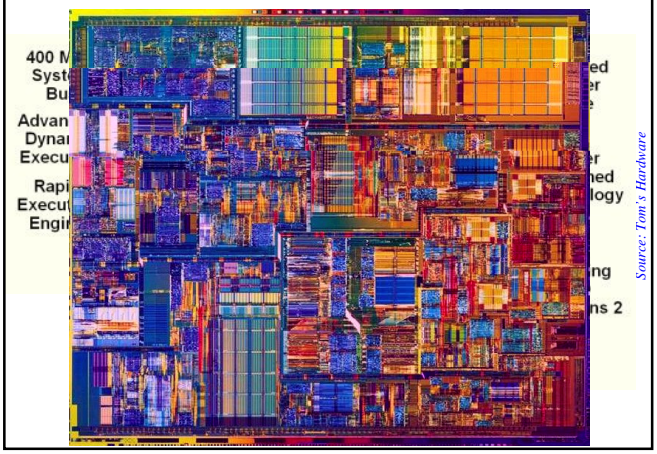


Source: Tom's Hardware

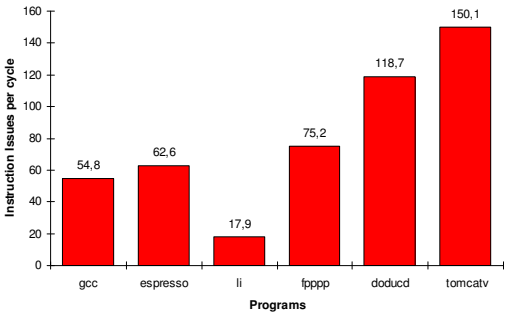
Trace Cache



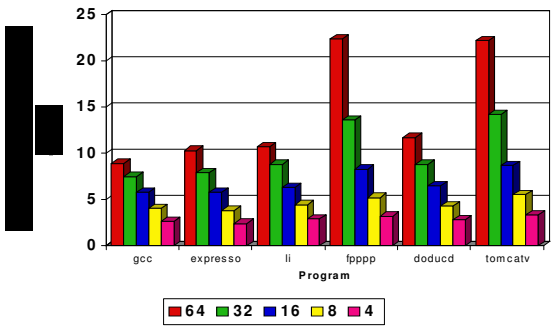
Pentium IV



Ideally...



Factoring in main constraints…



Nombre d'instructions/cycle

In reality…

- 4 instructions/cycle
- Max 2,6 instructions per cycle; 2 on average