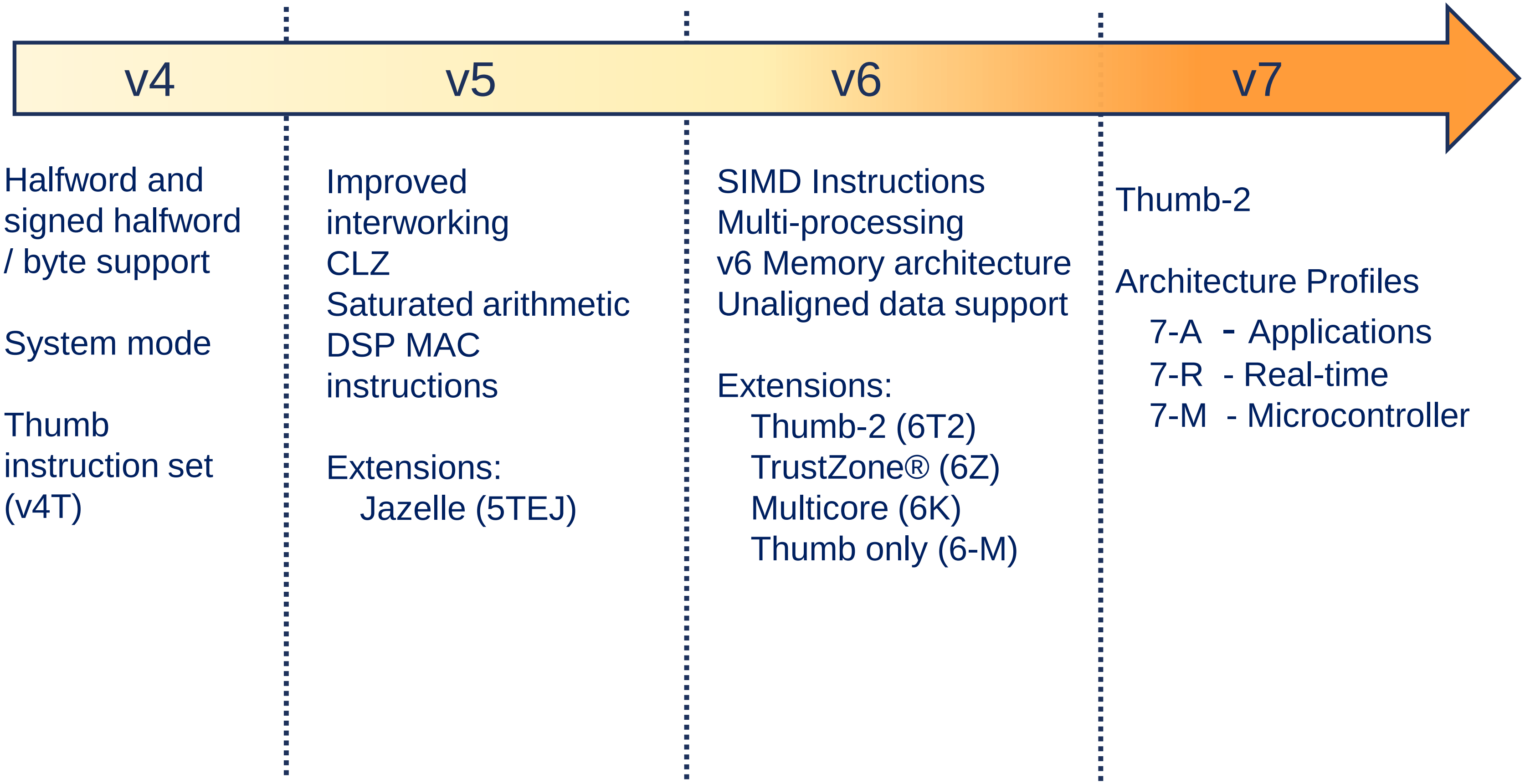
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Information Technology
and Electrical Engineering

ARM ISA Overview

Development of the ARM Architecture



v4

Halfword and signed halfword / byte support

System mode

Thumb instruction set (v4T)

v5

Improved interworking
CLZ
Saturated arithmetic
DSP MAC instructions

Extensions:
Jazelle (5TEJ)

v6

SIMD Instructions
Multi-processing
v6 Memory architecture
Unaligned data support

Extensions:
Thumb-2 (6T2)
TrustZone® (6Z)
Multicore (6K)
Thumb only (6-M)

v7

Thumb-2

Architecture Profiles
7-A - Applications
7-R - Real-time
7-M - Microcontroller

- **Note that implementations of the same architecture can be different**
  - Cortex-A8 - architecture v7-A, with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A, with an 8-stage pipeline

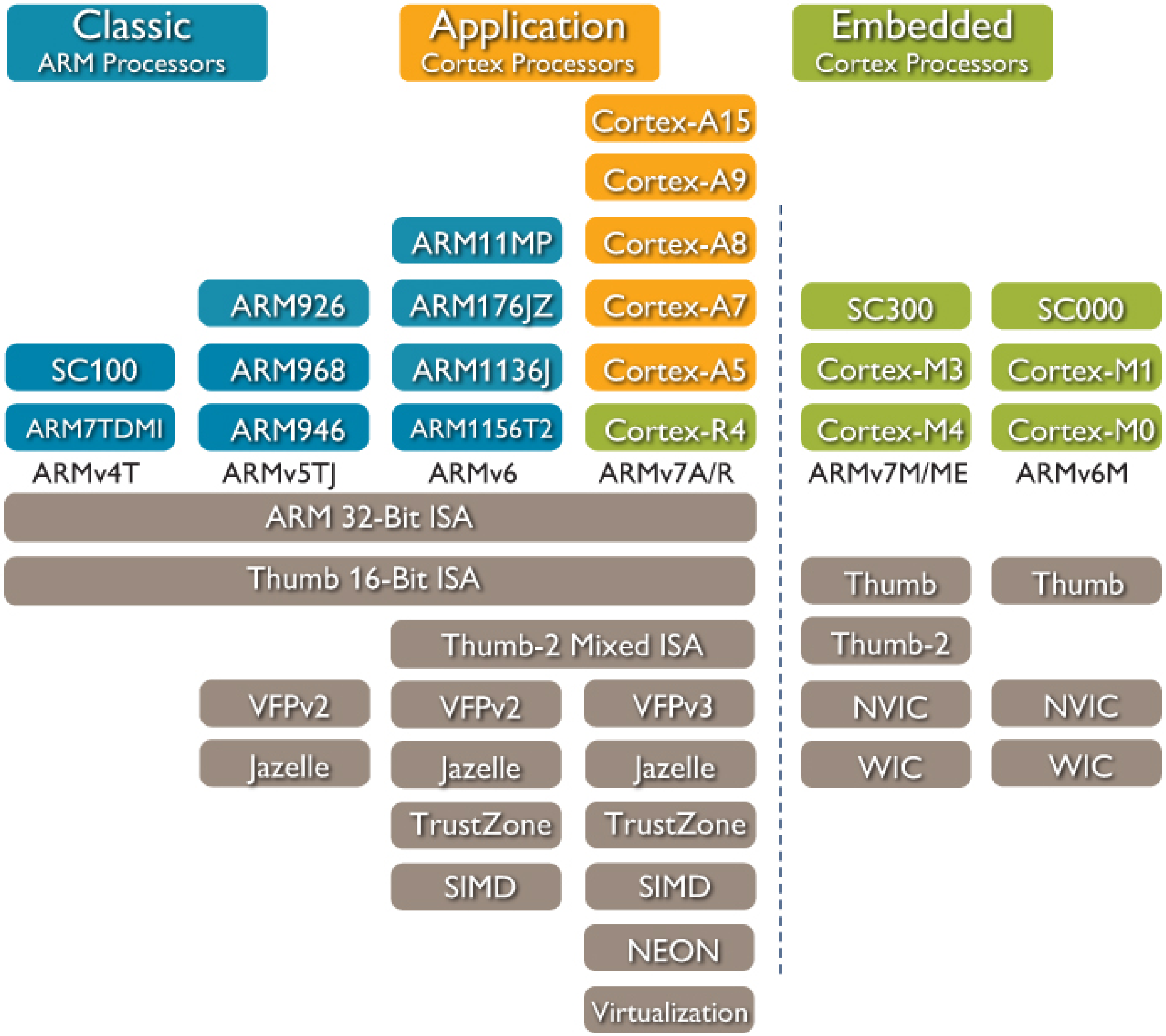
Architecture ARMv7 profiles

- **Application profile (ARMv7-A)**
  - Memory management support (MMU)
  - Highest performance at low power
    - Influenced by multi-tasking OS system requirements
  - TrustZone and Jazelle-RCT for a safe, extensible system
  - e.g. Cortex-A5, Cortex-A9

- **Real-time profile (ARMv7-R)**
  - Protected memory (MPU)
  - Low latency and predictability ‘real-time’ needs
  - Evolutionary path for traditional embedded business
  - e.g. Cortex-R4

- **Microcontroller profile (ARMv7-M, ARMv7E-M, ARMv6-M)**
  - Lowest gate count entry point
  - Deterministic and predictable behavior a key priority
  - Deeply embedded use
  - e.g. Cortex-M3

Which architecture is my processor?



Classic
ARM Processors

Application
Cortex Processors

Embedded
Cortex Processors

SC100
ARM7TDMI
ARMv4T

ARM926
ARM968
ARM946
ARMv5TJ
VFPv2
Jazelle

ARM11MP
ARM176JZ
ARM1136J
ARM1156T2
ARMv6
VFPv2
Jazelle
TrustZone
SIMD

Cortex-A15
Cortex-A9
Cortex-A8
Cortex-A7
Cortex-A5
Cortex-R4
ARMv7A/R
VFPv3
Jazelle
TrustZone
SIMD
NEON
Virtualization

ARM 32-Bit ISA

Thumb 16-Bit ISA

Thumb-2 Mixed ISA

SC300
Cortex-M3
Cortex-M4
ARMv7M/ME
Thumb
Thumb-2
NVIC
WIC

SC000
Cortex-M1
Cortex-M0
ARMv6M
Thumb
NVIC
WIC

Data Sizes and Instruction Sets

- **ARM is a 32-bit load / store RISC architecture**
  - The only memory accesses allowed are loads and stores
  - Most internal registers are 32 bits wide
  - Most instructions execute in a single cycle

- **When used in relation to ARM cores**
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
  - **Doubleword** means 64 bits (eight bytes)

- **ARM cores implement two basic instruction sets**
  - **ARM** instruction set – instructions are all 32 bits long
  - **Thumb** instruction set – instructions are a mix of 16 and 32 bits
    - Thumb-2 technology added many extra 32- and 16-bit instructions to the original 16-bit Thumb instruction set

- **Depending on the core, may also implement other instruction sets**
  - **VFP** instruction set – 32 bit (vector) floating point instructions
  - **NEON** instruction set – 32 bit SIMD instructions
  - **Jazelle-DBX** - provides acceleration for Java VMs (with additional software support)
  - **Jazelle-RCT** - provides support for interpreted languages

ARMv7-M Profile Overview

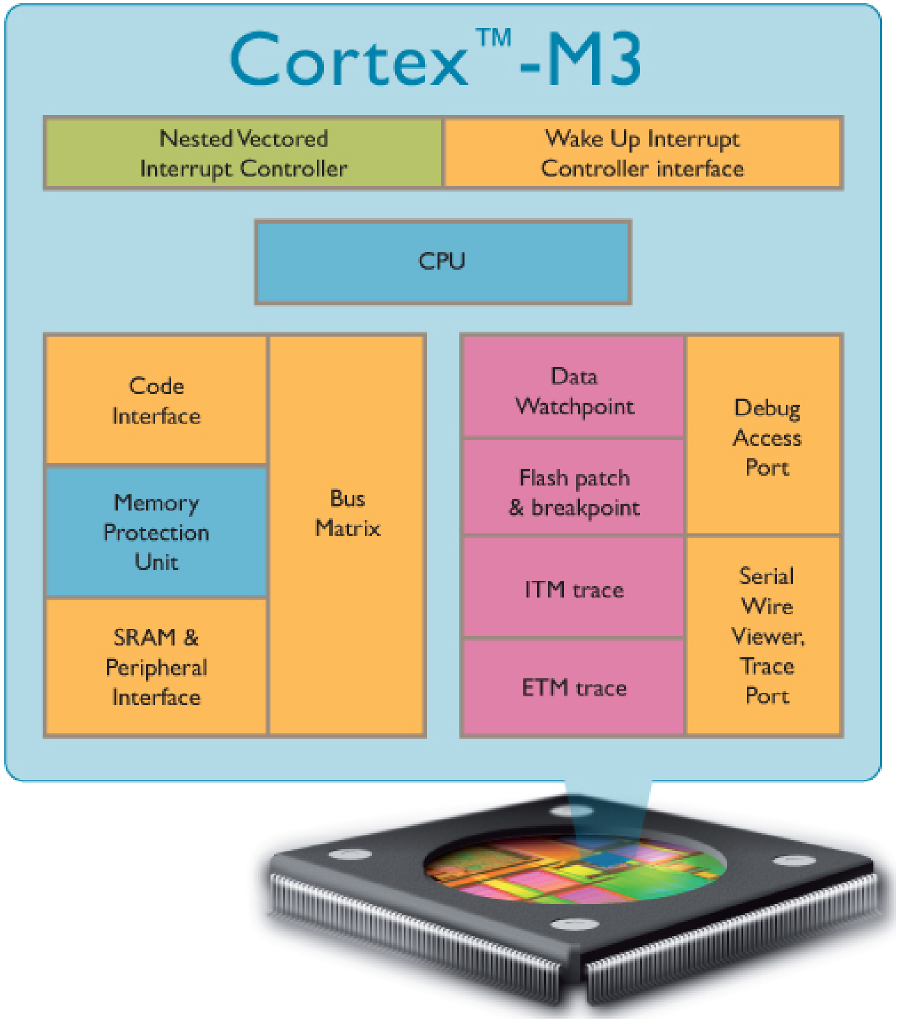
- **v7-M Cores are designed to support the microcontroller market**
  - Simpler to program – entire application can be programmed in C
  - Fewer features needed than in application processors

- **Register and ISA changes from other ARM cores**
  - No ARM instruction set support
  - Only one set of registers
  - xPSR has different bits than CPSR

- **Different modes and exception models**
  - Only two modes: Thread mode and Handler mode
  - Vector table is addresses, not instructions
  - Exceptions automatically save state (r0-r3, r12, lr, xPSR, pc) on the stack

- **Different system control/memory layout**
  - Cores have a fixed memory map
  - No coprocessor 15 – controlled through memory mapped control registers

Cortex-M3



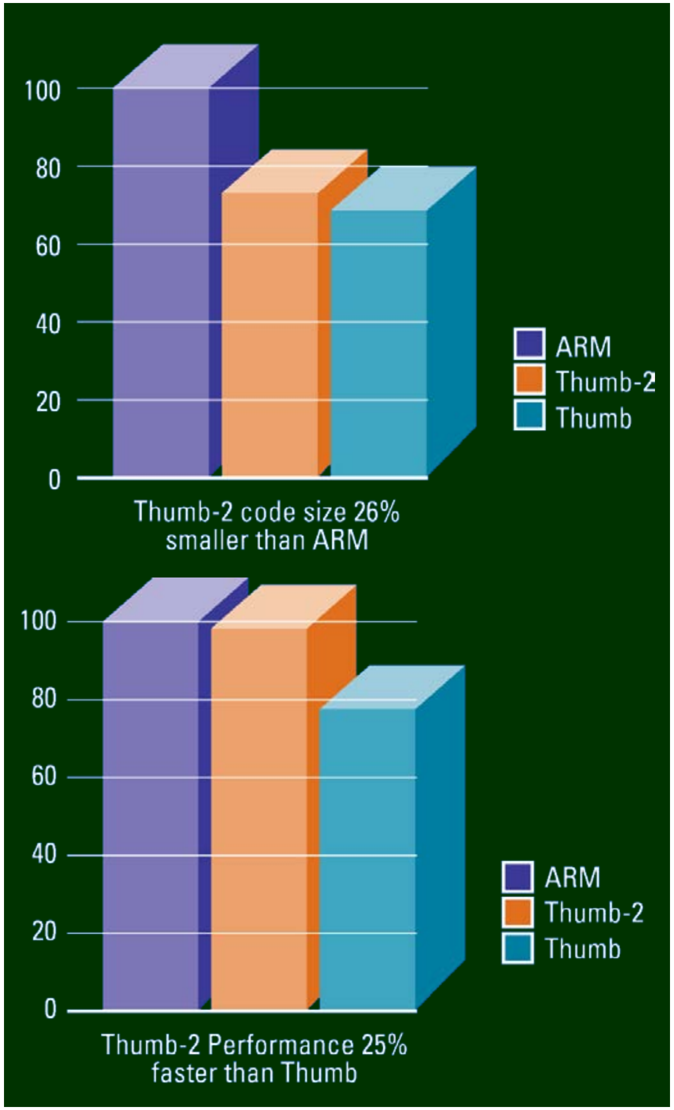
- **ARMv7-M Architecture**
  - Thumb-2 only
- **Fully programmable in C**
- **3-stage pipeline**
- **von Neumann architecture**
- **Optional MPU**
- **AHB-Lite bus interface**
- **Fixed memory map**
- **1-240 interrupts**
  - Configurable priority levels
  - Non-Maskable Interrupt support
  - Debug and Sleep control
- **Serial wire or JTAG debug**
- **Optional ETM**

The Thumb-2 instruction set

- **Variable-length instructions**
  - ARM instructions are a fixed length of 32 bits
  - Thumb instructions are a fixed length of 16 bits
  - Thumb-2 instructions can be either 16-bit or 32-bit

- **Thumb-2 gives approximately 26% improvement in code density over ARM**

- **Thumb-2 gives approximately 25% improvement in performance over Thumb**



Processor Modes

- The ARM has seven basic operating modes:
  - Each mode has access to:
    - Its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

	Mode	Description	
Exception modes	**Supervisor (SVC)**	Entered on reset and when a Software Interrupt instruction (SWI) is executed	**Privileged modes**
	FIQ	Entered when a high priority (fast) interrupt is raised	
	IRQ	Entered when a low priority (normal) interrupt is raised	
	Abort	Used to handle memory access violations	
	Undef	Used to handle undefined instructions	
	System	Privileged mode using the same registers as User mode	
	User	Mode under which most Applications / OS tasks run	**Unprivileged mode**

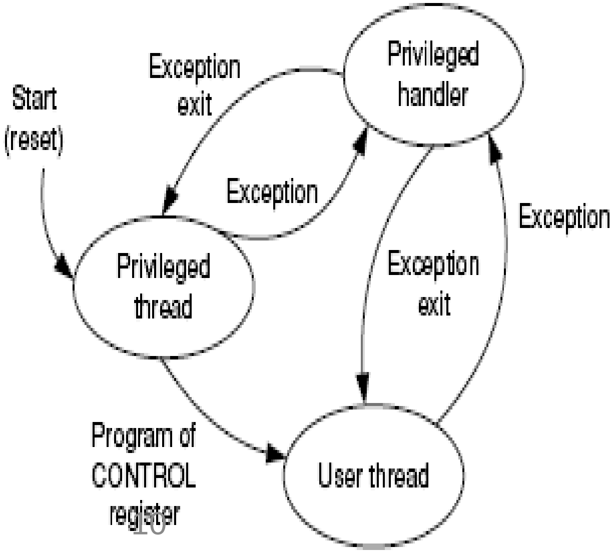
Operating Modes

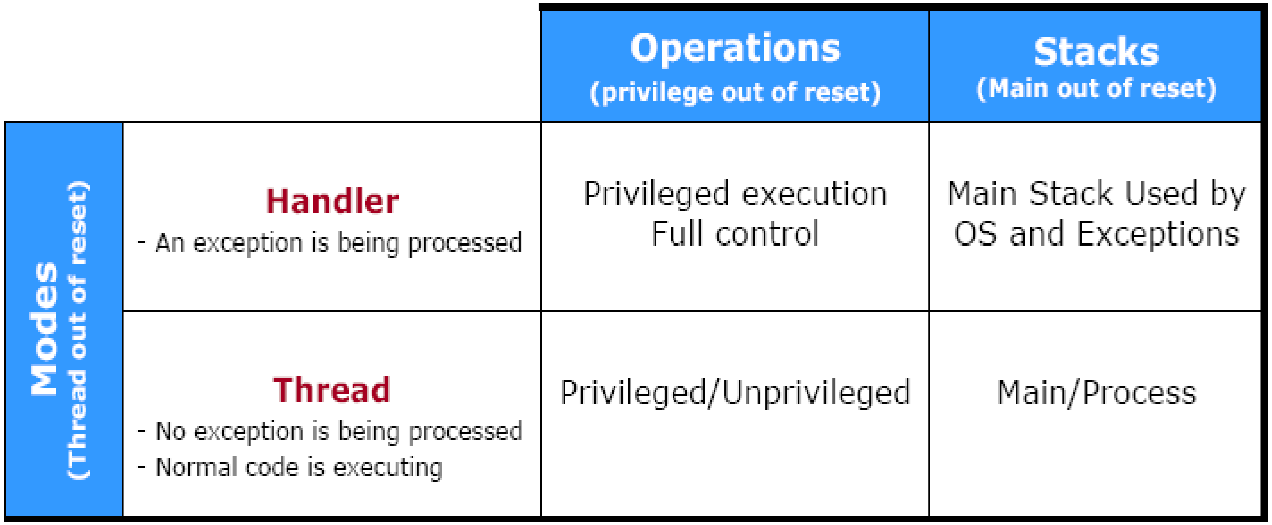
User mode:

- Normal program execution mode
- System resources unavailable
- Mode changed by exception only

Exception modes:

- Entered upon exception
- Full access to system resources
- Mode changed freely





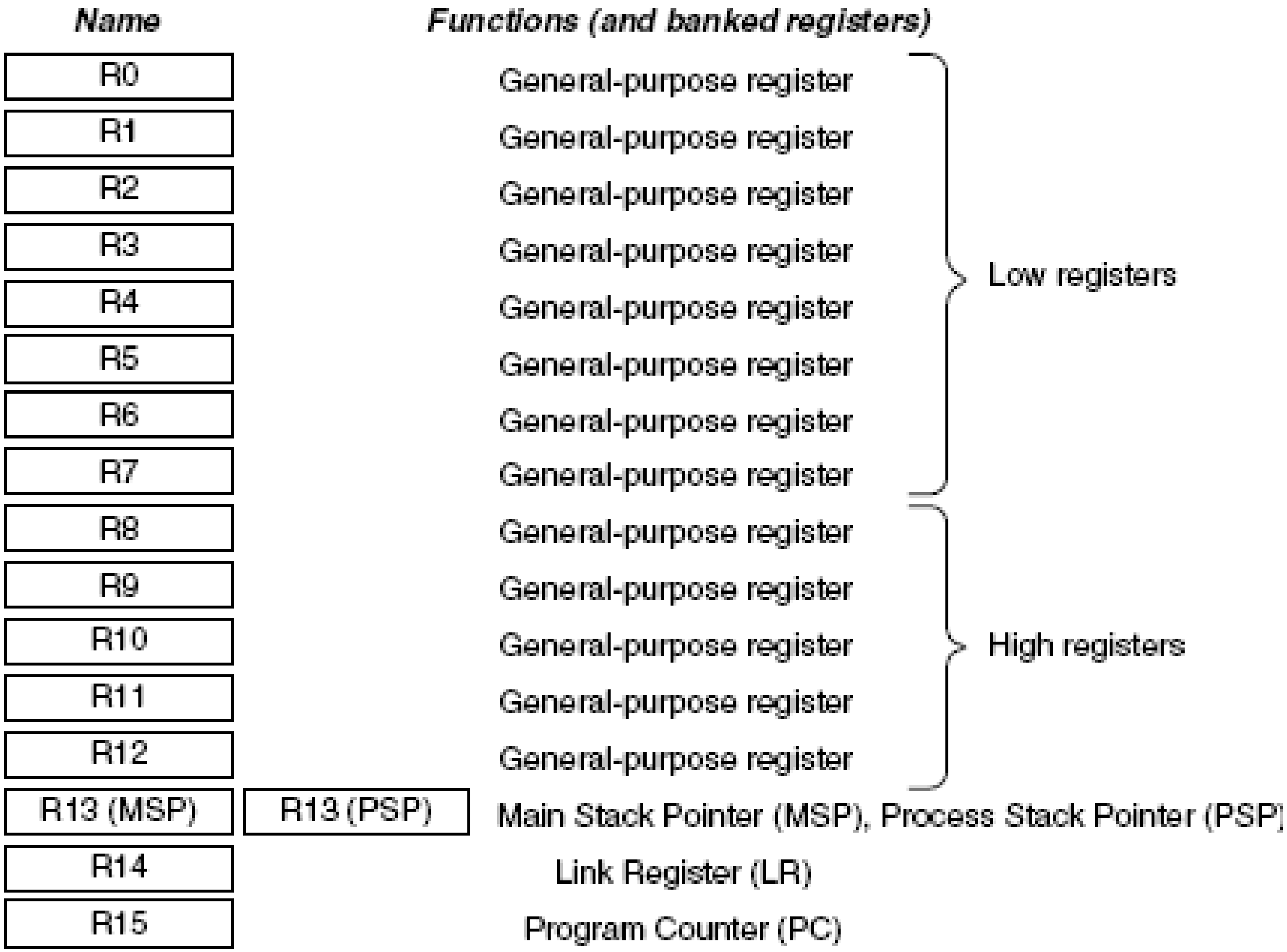
Modes (Thread out of reset)		Operations (privilege out of reset)	Stacks (Main out of reset)
	Handler - An exception is being processed	Privileged execution Full control	Main Stack Used by OS and Exceptions
	Thread - No exception is being processed - Normal code is executing	Privileged/Unprivileged	Main/Process

Exceptions

Exception	Mode	Priority	IV Address
Reset	Supervisor	1	0x00000000
Undefined instruction	Undefined	6	0x00000004
Software interrupt	Supervisor	6	0x00000008
Prefetch Abort	Abort	5	0x0000000C
Data Abort	Abort	2	0x00000010
Interrupt	IRQ	4	0x00000018
Fast interrupt	FIQ	3	0x0000001C

Table 1 - Exception types, sorted by Interrupt Vector addresses

Registers



Name		*Functions (and banked registers)*	
R0		General-purpose register	Low registers
R1		General-purpose register	
R2		General-purpose register	
R3		General-purpose register	
R4		General-purpose register	
R5		General-purpose register	
R6		General-purpose register	
R7		General-purpose register	
R8		General-purpose register	High registers
R9		General-purpose register	
R10		General-purpose register	
R11		General-purpose register	
R12		General-purpose register	
R13 (MSP)	R13 (PSP)	Main Stack Pointer (MSP), Process Stack Pointer (PSP)	
R14		Link Register (LR)	
R15		Program Counter (PC)	

ARM Registers

- 31 general-purpose 32-bit registers
- 16 visible, R0 – R15
- Others speed up the exception process

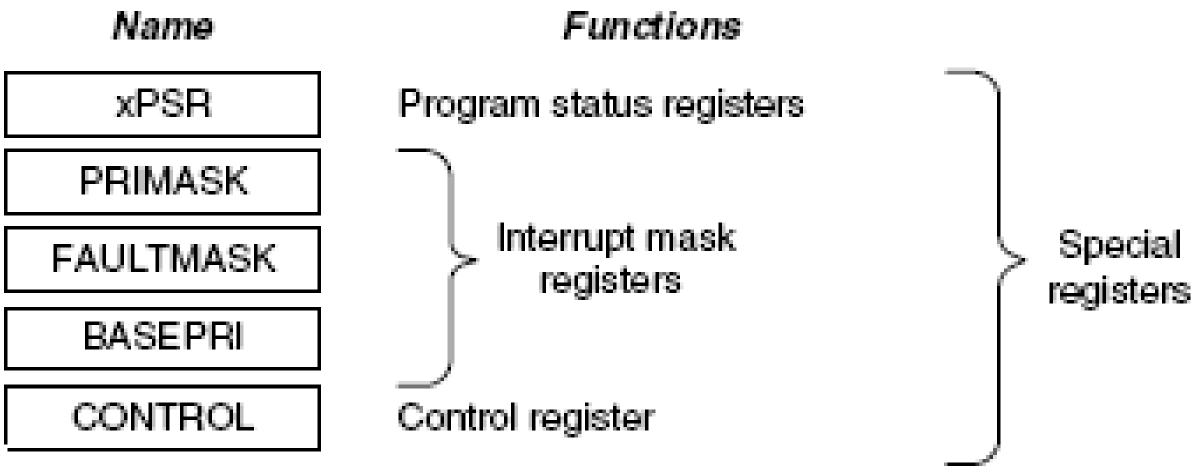
ARM Registers (2)

- Special roles:
  - Hardware
    - R14 – Link Register (LR):
      optionally holds return address
      for branch instructions
    - R15 – Program Counter (PC)
  - Software
    - R13 - Stack Pointer (SP)

ARM Registers (3)

- Current Program Status Register (CPSR)
- Saved Program Status Register (SPSR)
- On exception, entering *mod* mode:
  - (PC + 4) → LR
  - CPSR → SPSR_mod
  - PC ← IV address
  - R13, R14 replaced by R13_mod, R14_mod
  - In case of FIQ mode R7 – R12 also replaced

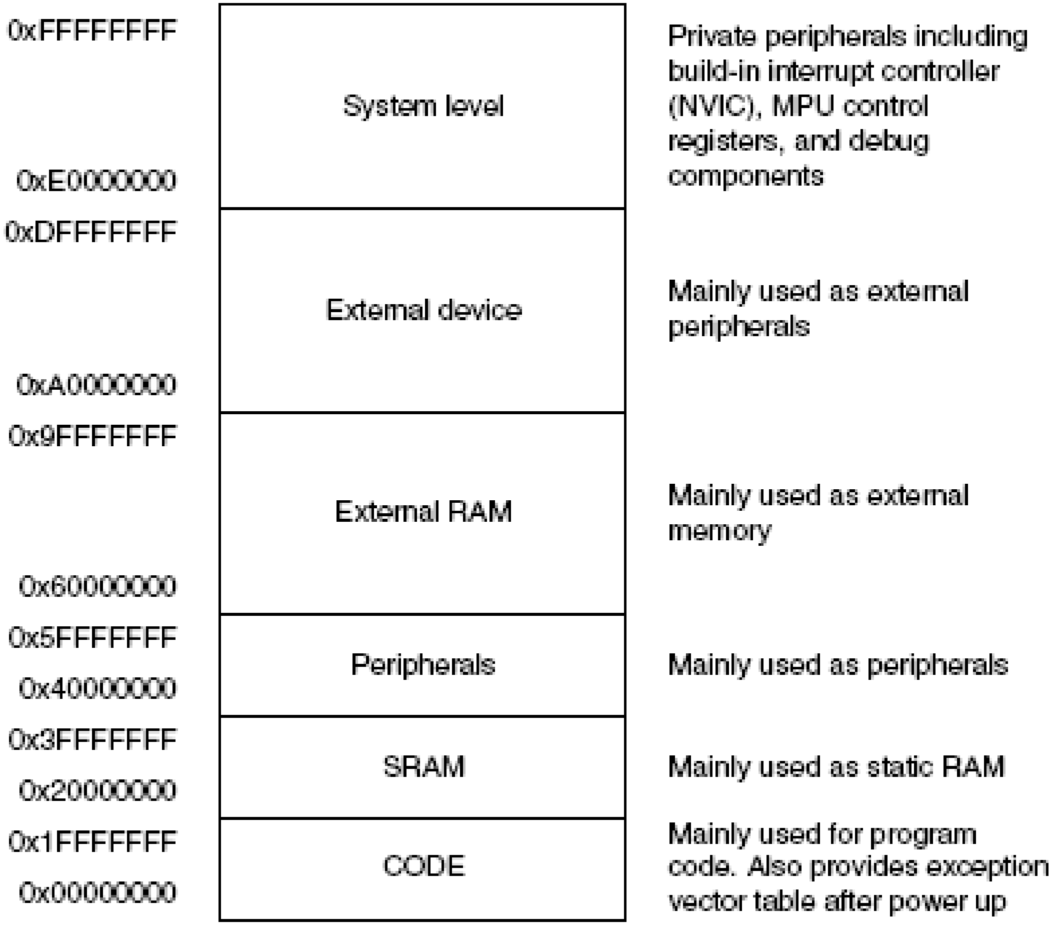
Special Registers



Register	Function
xPSR	Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

Memory map

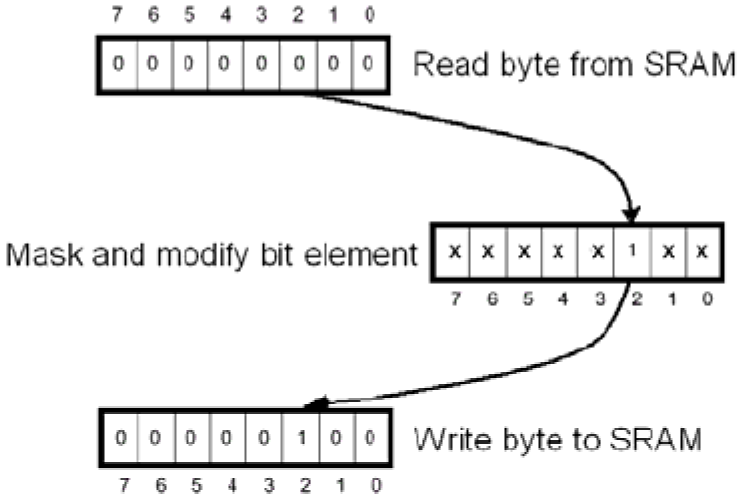
- Statically defined memory map (faster addr decoding) 4GB of address psace



0xFFFFFFFF 0xE0000000	System level	Private peripherals including build-in interrupt controller (NVIC), MPU control registers, and debug components
0xDFFFFFFF 0xA0000000	External device	Mainly used as external peripherals
0x9FFFFFFF 0x60000000	External RAM	Mainly used as external memory
0x5FFFFFFF 0x40000000	Peripherals	Mainly used as peripherals
0x3FFFFFFF 0x20000000	SRAM	Mainly used as static RAM
0x1FFFFFFF 0x00000000	CODE	Mainly used for program code. Also provides exception vector table after power up

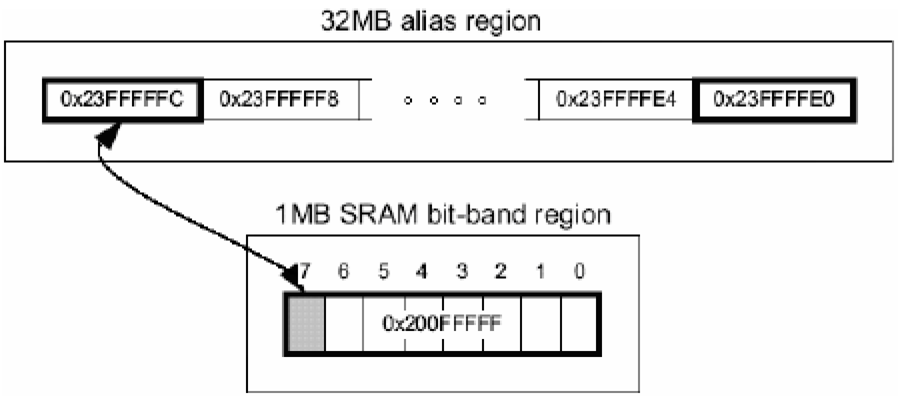
Bit Banding

- Fast single-bit manipulation: 1MB → 32MB aliased regions in SRAM & Peripheral space



```
LDR R0,=0x200FFFFF ; Setup address
MOV R2, #0x4       ; Setup data
LDR R1, [R0]       ; Read
ORR R1, R2         ; Modify bit
STR R1, [R0]       ; Write back result
```

Traditional bit manipulation method

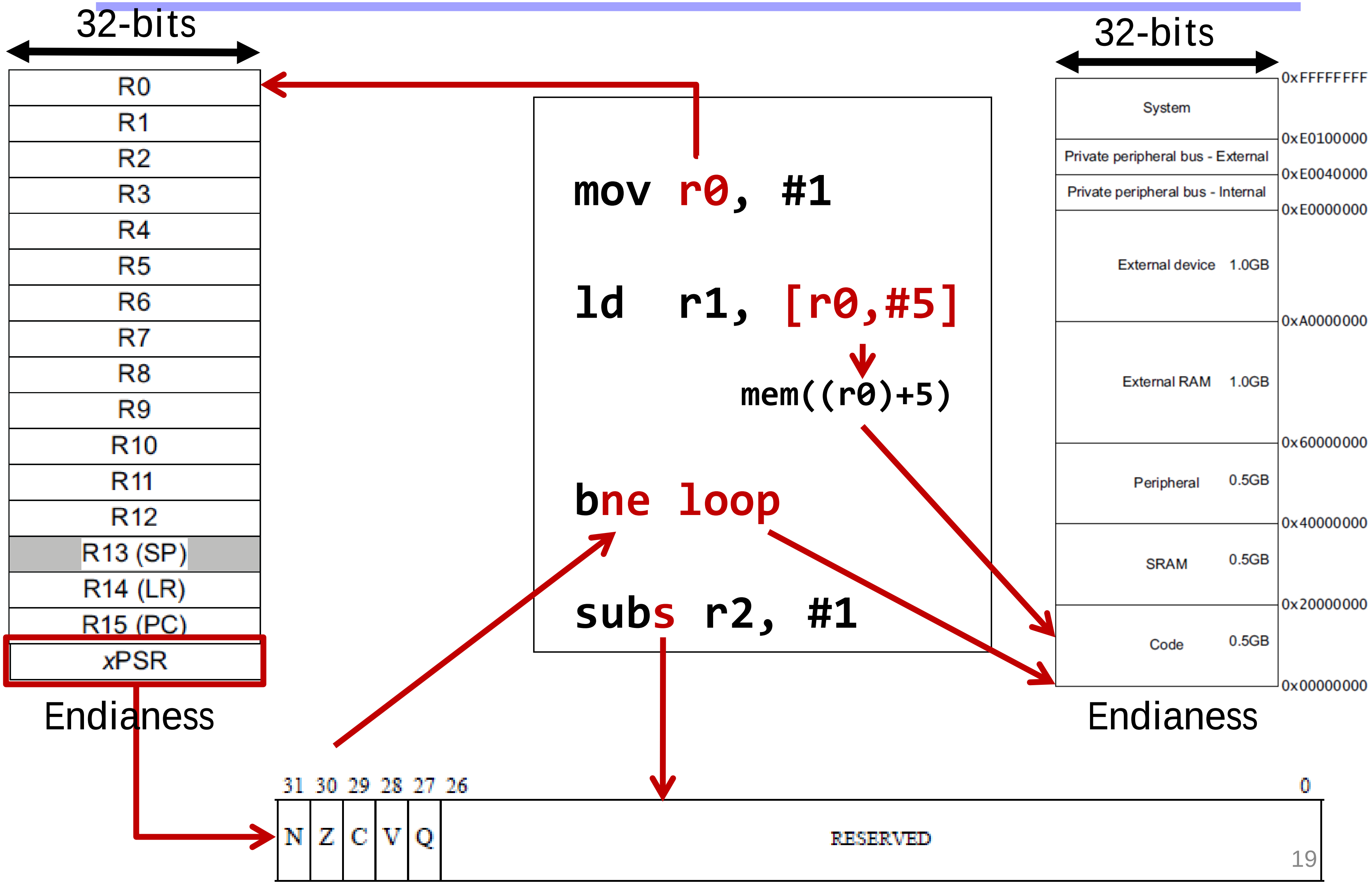


```
LDR R0,=0x23FFFFFC ; Setup address
MOV R1, #0x1       ; Setup data
STR R1, [R0]       ; Write
```

Direct, single cycle access with bit banding

Major Elements of ISA

(registers, memory, word size, endianess, conditions, instructions, addressing modes)



32-bits

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)
*x*PSR

Endianess

```
mov r0, #1

ld  r1, [r0,#5]
             mem((r0)+5)

bne loop

subs r2, #1
```

32-bits

Region	Size	Address
		0xFFFFFFFF
System		0xE0100000
Private peripheral bus - External		0xE0040000
Private peripheral bus - Internal		0xE0000000
External device	1.0GB	0xA0000000
External RAM	1.0GB	0x60000000
Peripheral	0.5GB	0x40000000
SRAM	0.5GB	0x20000000
Code	0.5GB	0x00000000

Endianess

31	30	29	28	27	26 … 0
N	Z	C	V	Q	RESERVED

Traditional ARM instructions

- Fixed length of 32 bits
- Commonly take two or three operands
- Process data held in registers
- Shift & ALU operation in single clock cycle
- Access memory with load and store instructions only
  - Load/Store multiple register
- Can be extended to execute conditionally by adding the appropriate suffix
- Affect the CPSR status flags by adding the ‘S’ suffix to the instruction

Thumb-2

- Original 16-bit Thumb instruction set
  - a subset of the full ARM instructions
  - performs similar functions to selective 32-bit ARM instructions but in 16-bit code size
- For ARM instructions that are not available
  - more 16-bit Thumb instructions are needed to execute the same function compared to using ARM instructions
  - but performance may be degraded
- Hence the introduction of the Thumb-2 instruction set
  - enhances the 16-bit Thumb instructions with additional 32-bit instructions
- All ARMv7 chips support the Thumb-2 (& ARM) instruction set
  - but Cortex-M3 supports only the 16-bit/32-bit Thumb-2 instruction set

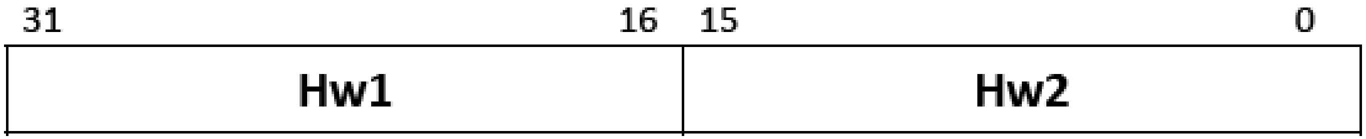
16bit Thumb-2

Some of the changes used to reduce the length of the instructions from 32 bits to 16 bits:

- reduce the number of bits used to identify the register
  - less number of registers can be used
- reduce the number of bits used for the immediate value
  - smaller number range
- remove options such as 'S'
  - make it default for some instructions
- remove conditional fields (N, Z, V, C)
- no conditional executions (except branch)
- remove the optional shift (and no barrel shifter operation
  - introduce dedicated shift instructions
- remove some of the instructions
  - more restricted coding

Thumb-2 Implementation

- The 32-bit ARM Thumb-2 instructions are added through the space occupied by the Thumb BL and BLX instructions



31 16 15 0

Hw1 Hw2

32-bit Thumb-2 Instruction format

- The first Halfword (Hw1)
  - determines the instruction length and functionality
- If the processor decodes the instruction as 32-bit long
  - the processor fetches the second halfword (hw2) of the instruction from the instruction address plus two

Unified Assembly Language

- UAL supports generation of either Thumb-2 or ARM instructions from the same source code
  - same syntax for both the Thumb code and ARM code
  - enable portability of code for different ARM processor families
- Interpretation of code type is based on the directive listed in the assembly file
- Example:
  - For GNU GAS, the directive for UAL is

.syntax unified

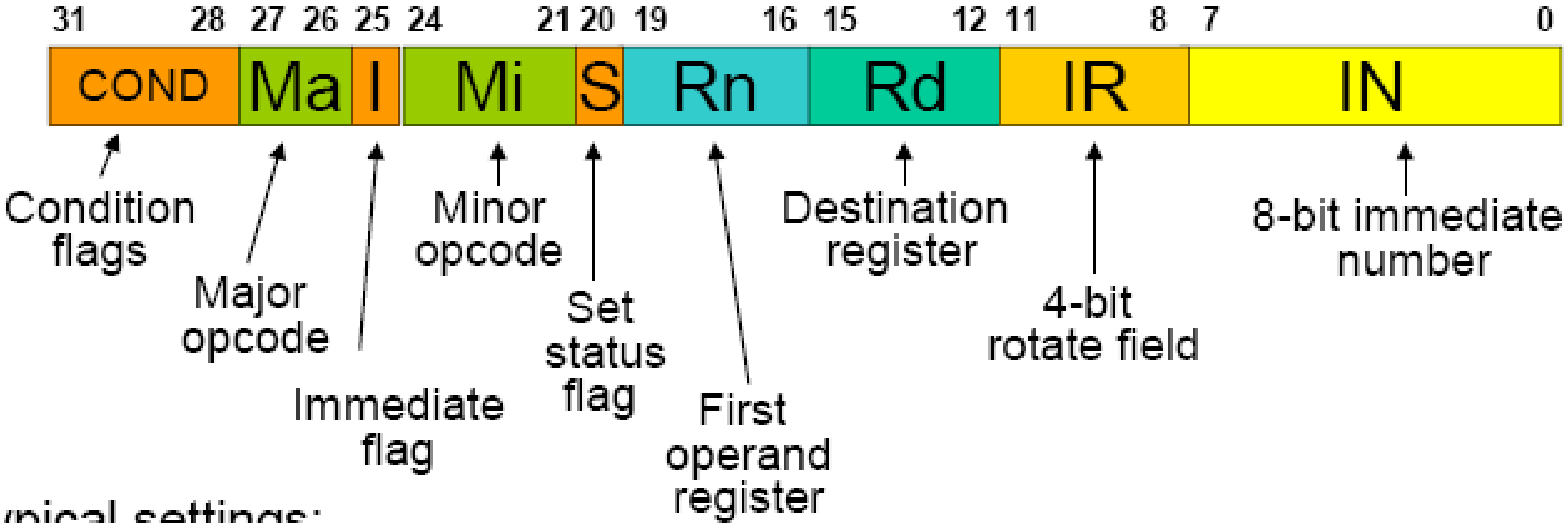
- For ARM assembler, the directive for UAL is

THUMB

32bit Instruction Encoding

Example: ADD instruction format

- ARM 32-bit encoding for ADD with immediate field



31 28 27 26 25 24 21 20 19 16 15 12 11 8 7 0

COND Ma I Mi S Rn Rd IR IN

Condition flags

Major opcode

Immediate flag

Minor opcode

Set status flag

First operand register

Destination register

4-bit rotate field

8-bit immediate number

Typical settings:

Major opcode = 00 (this indicates data operation instructions)

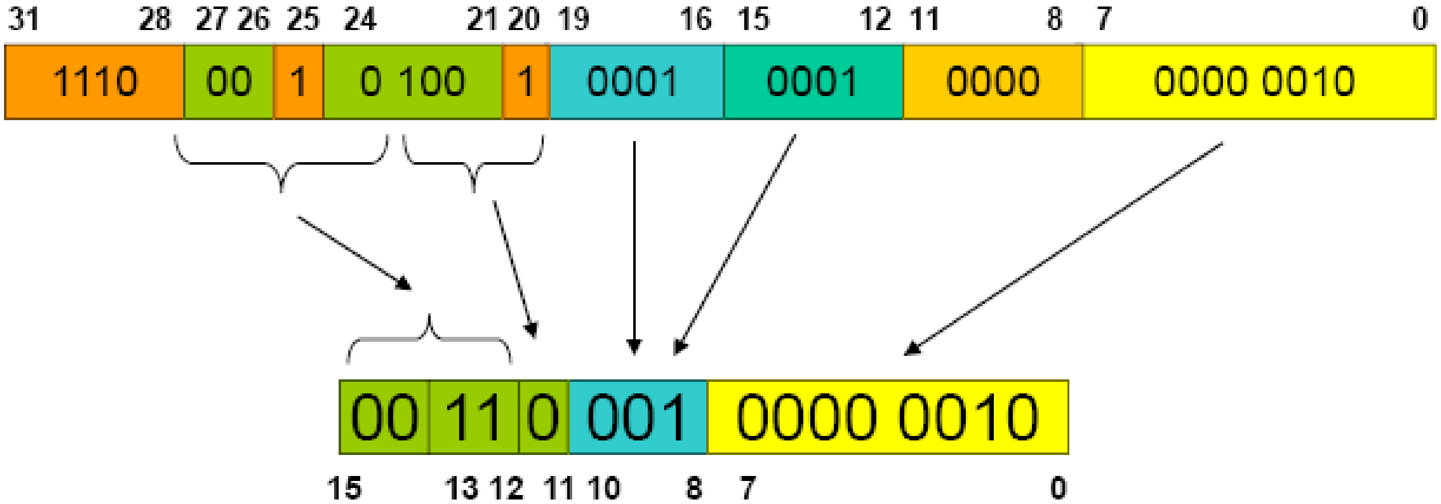
Minor opcode = 0100 (specifically, 100 ⇨ ADD instruction)

Immediate flag = 1 (immediate field in operand 2)

Set status flag = 1 (set carry flag after operation)

ARM and 16-bit Instruction Encoding

ARM 32-bit encoding: ADDS `r1, r1, #2`



31 28 27 26 25 24 21 20 19 16 15 12 11 8 7 0

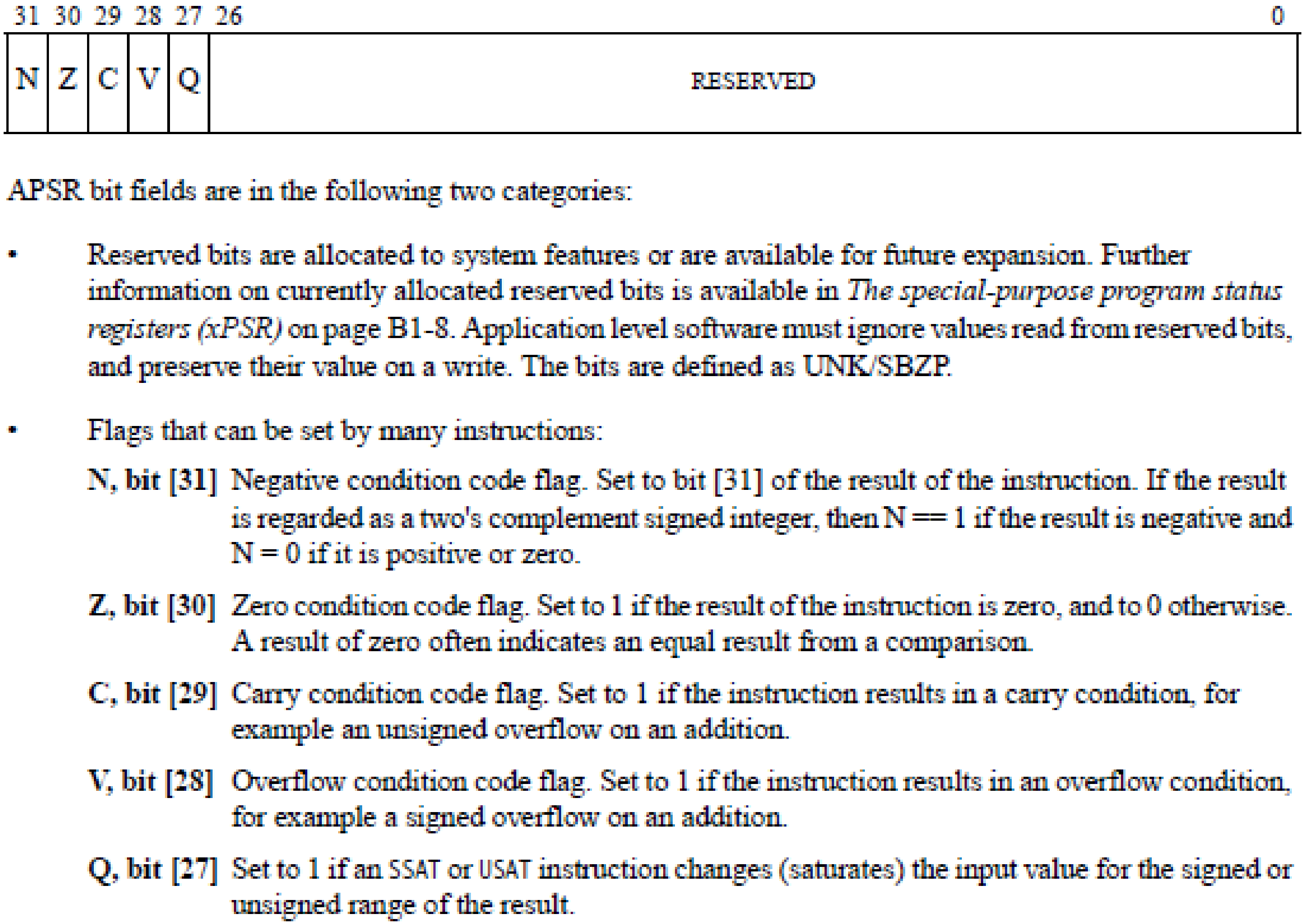
1110 | 00 | 1 | 0 100 | 1 | 0001 | 0001 | 0000 | 0000 0010

00 | 11 | 0 | 001 | 0000 0010

15 13 12 11 10 8 7 0

- Equivalent 16-bit Thumb instruction: ADD `r1, #2`
  - No condition flag
  - No rotate field for the immediate number
  - Use 3-bit encoding for the register
  - Shorter opcode with implicit flag settings (e.g. the set status flag is always set)

Application Program Status Register (APSR)



31	30	29	28	27	26 … 0
N	Z	C	V	Q	RESERVED

APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.
- Flags that can be set by many instructions:

  N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.

  Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  Q, bit [27] Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.

Updating the APSR

- SUB Rx, Ry
  - Rx = Rx - Ry
  - APSR unchanged
- SUBS
  - Rx = Rx - Ry
  - APSR N or Z bits might be set
- ADD Rx, Ry
  - Rx = Rx + Ry
  - APSR unchanged
- ADDS
  - Rx = Rx + Ry
  - APSR C or V bits might be set

Conditional Execution

- Each data processing instruction prefixed by condition code
- Result – smooth flow of instructions through pipeline
- 16 condition codes:

EQ	equal	MI	negative	HI	unsigned higher	GT	signed greater than
NE	not equal	PL	positive or zero	LS	unsigned lower or same	LE	signed less than or equal
CS	unsigned higher or same	VS	overflow	GE	signed greater than or equal	AL	always
CC	unsigned lower	VC	no overflow	LT	signed less than	NV	special purpose

Conditional Execution

- Every ARM (32 bit) instruction is conditionally executed.
- The top four bits are ANDed with the CPSR condition codes, If they do not matched the instruction is executed as NOP
- The AL condition is used to execute the instruction irrespective of the value of the condition code flags.
- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". Ex: SUBS r1,r1,#1
- Conditional Execution improves code density and performance by reducing the number of forward branch instructions.

```
Normal                        Conditional
  CMP   r3,#0                          CMP   r3,#0
  BEQ   skip                           ADDNE r0,r1,r2
  ADD   r0,r1,r2
skip
```

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by post-fixing them with the appropriate condition code
  - This can increase code density and increase performance by reducing the number of forward branches

```
CMP      r0, r1         ←  r0 - r1, compare r0 with r1 and set flags
ADDGT    r2, r2, #1     ←  if > r2=r2+1 flags remain unchanged
ADDLE    r3, r3, #1     ←  if <= r3=r3+1 flags remain unchanged
```

- By default, data processing instructions do not affect the condition flags but this can be achieved by post fixing the instruction (and any condition code) with an “S”

```
loop
    ADD      r2, r2, r3       ←  r2=r2+r3
    SUBS     r1, r1, #0x01    ←  decrement r1 and set flags
    BNE      loop             ←  if Z flag clear then branch
```

Conditional execution examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

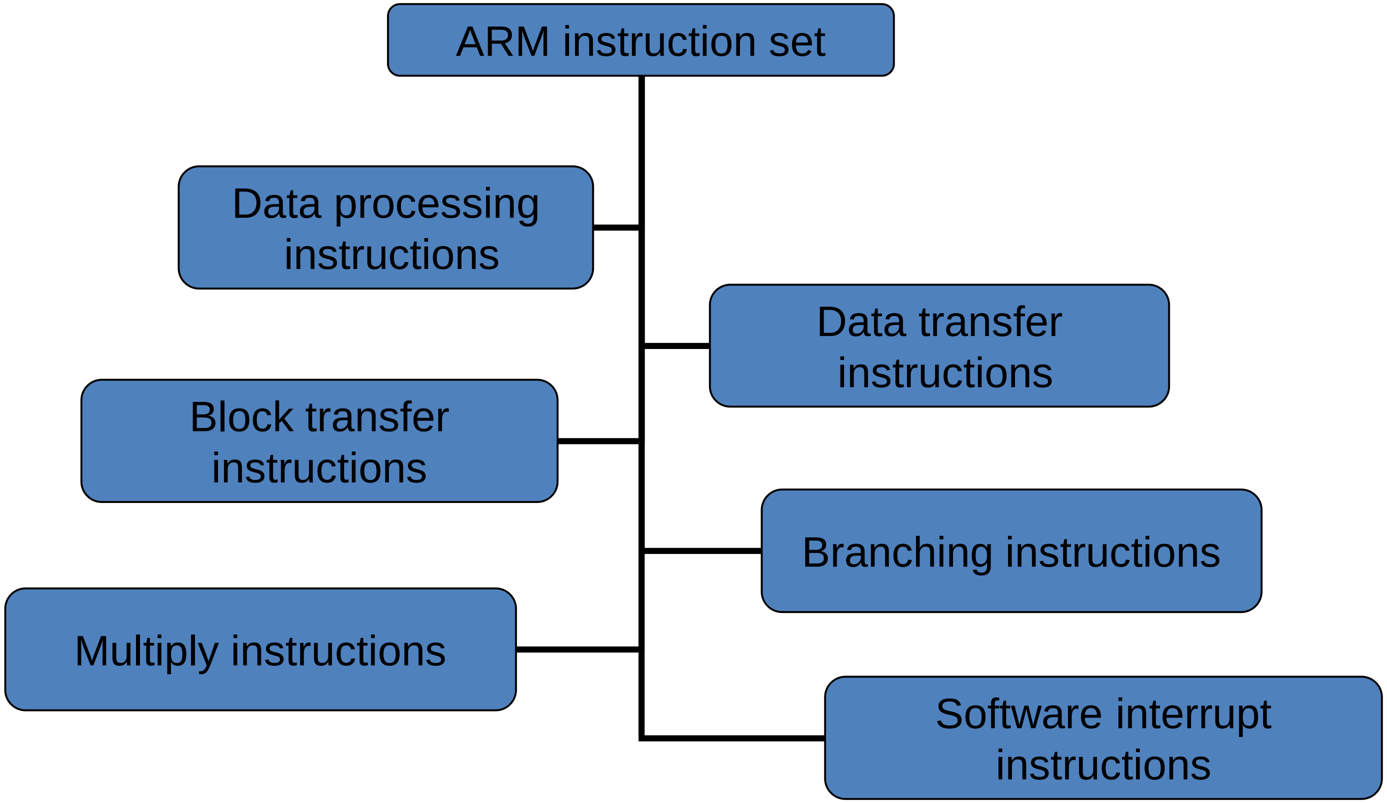
- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
  CMP r0, #0
  ADDEQ r1, r1,
#1
  ADDNE r2, r2,
#1
  ...
```

- 3 instructions
- 3 words
- 3 cycles

ARM Instruction Set (3)



ARM instruction set

Data processing instructions

Data transfer instructions

Block transfer instructions

Branching instructions

Multiply instructions

Software interrupt instructions

Data Processing Instructions

- Arithmetic and logical operations
- 3-address format:
  - Two 32-bit operands
    (op1 is register, op2 is register or immediate)
  - 32-bit result placed in a register
- Barrel shifter for op2 allows full 32-bit shift within instruction cycle

Data Processing Instructions (2)

- Arithmetic operations:
  - ADD, ADDC, SUB, SUBC, RSB, RSC
- Bit-wise logical operations:
  - AND, EOR, ORR, BIC
- Register movement operations:
  - MOV, MVN
- Comparison operations:
  - TST, TEQ, CMP, CMN

Data Processing Instructions (3)

Conditional codes

+

Data processing instructions

+

Barrel shifter

=

Powerful tools for efficient coded programs

Data Processing Instructions (4)

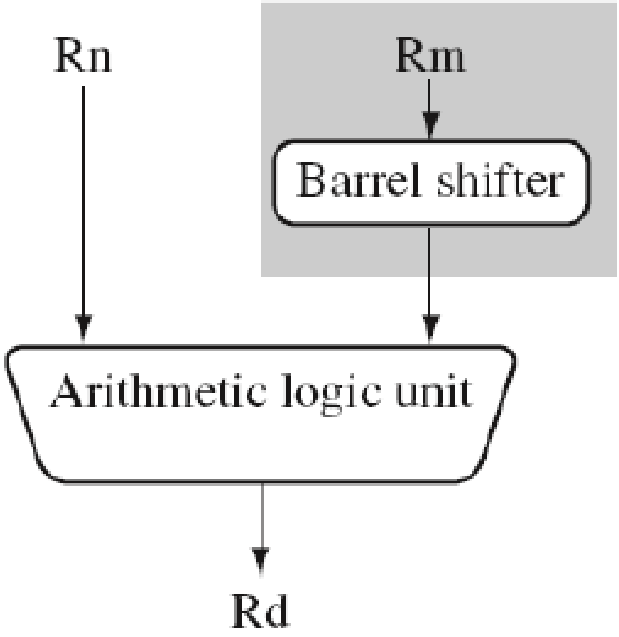
e.g.:

if (z==1) R1=R2+(R3*4)

compiles to

EQADDS R1,R2,R3, LSL #2

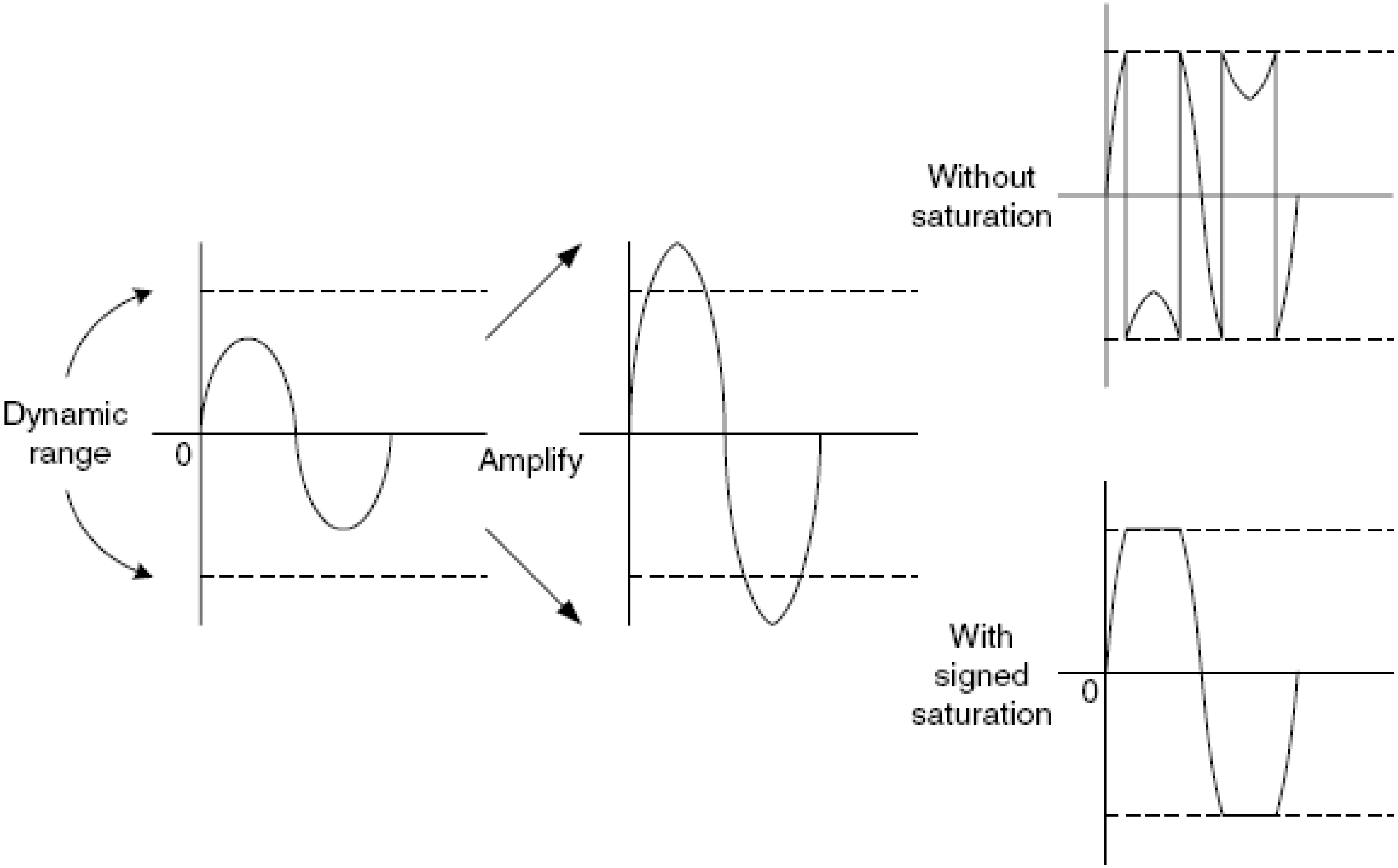
(SINGLE INSTRUCTION !)



Multiply Instructions

- Integer multiplication (32-bit result)
- Long integer multiplication (64-bit result)
- Built in Multiply Accumulate Unit (MAC)
- Multiply and accumulate instructions add product to running total

Saturated Arithmetic



Multiply Instructions

- Instructions:

MUL	Multiply	32-bit result
MULA	Multiply accumulate	32-bit result
UMULL	Unsigned multiply	64-bit result
UMLAL	Unsigned multiply accumulate	64-bit result
SMULL	Signed multiply	64-bit result
SMLAL	Signed multiply accumulate	64-bit result

Data Transfer Instructions

- Load/store instructions
- Used to move signed and unsigned
  Word, Half Word and Byte to and from registers
- Can be used to load PC
  (if target address is beyond branch instruction range)

LDR	Load Word	STR	Store Word
LDRH	Load Half Word	STRH	Store Half Word
LDRSH	Load Signed Half Word	STRSH	Store Signed Half Word
LDRB	Load Byte	STRB	Store Byte
LDRSB	Load Signed Byte	STRSB	Store Signed Byte

Addressing Modes

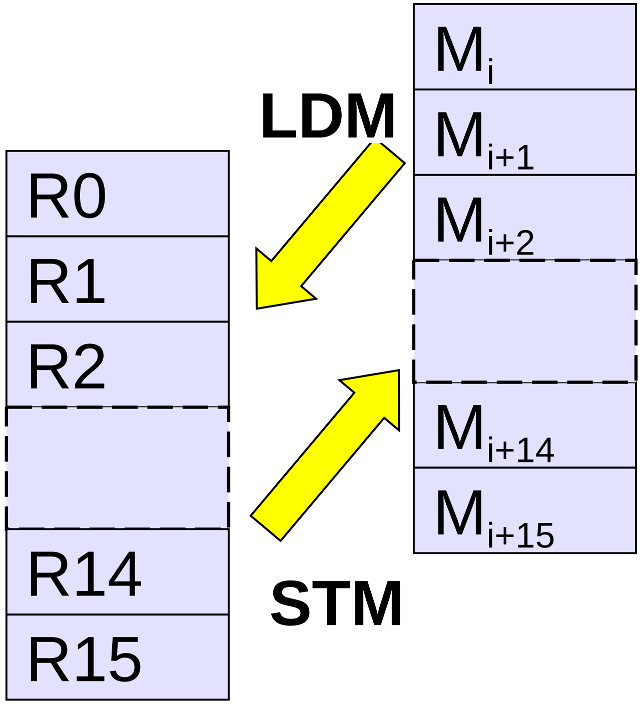
- Offset Addressing
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - [<Rn>, <offset>]
- Pre-indexed Addressing
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - [<Rn>, <offset>]!
- Post-indexed Addressing
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - [<Rn>], <offset>

<offset> options

- An immediate constant
  - #10
- An index register
  - <Rm>
- A shifted index register
  - <Rm>, LSL #<shift>

Block Transfer Instructions

- Load/Store Multiple instructions (*LDM/STM*)
- Whole register bank or a subset copied to memory or restored with single instruction

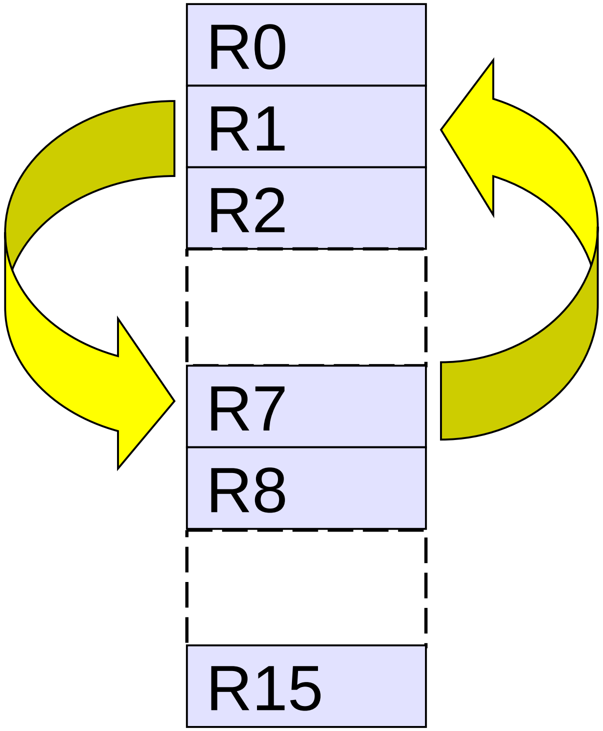


Swap Instruction

- Exchanges a word between registers
  - Two cycles

    but

    single atomic action
  - Support for RT semaphores



Branching Instructions

- *Branch* (B):

  jumps forwards/backwards up to 32 MB

- *Branch link* (BL):

  same + saves (PC+4) in LR

- Suitable for function call/return
- Condition codes for conditional branches

IF-THEN Instruction

- Another alternative to execute conditional code is the new 16-bit IF-THEN (IT) instruction
  - no change in program flow
  - no branching overhead
- Can use with 32-bit Thumb-2 instructions that do not support the ‘S’ suffix
- Example:

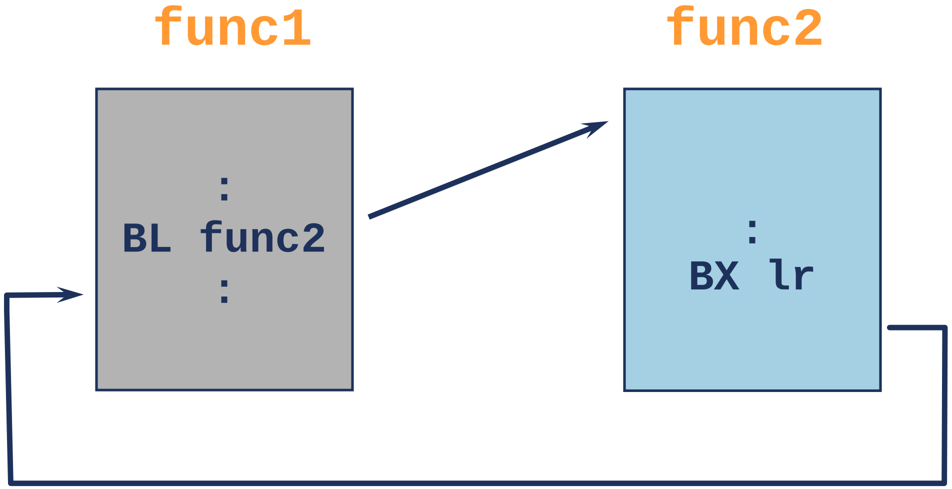
```
CMP R1, R2          ; If R1 = R2
IT EQ               ; execute next (1st)
                    ; instruction
ADDEQ R2, R1, R0    ; 1st instruction
```

- The conditional codes can be extended up to 4 instructions

Subroutines

- **Implementing a conventional subroutine call requires two steps**
  - Store the return address
  - Branch to the address of the required subroutine
- **These steps are carried out in one instruction, `BL`**
  - The return address is stored in the link register (`lr/r14`)
  - Branch to an address (range dependent on instruction set and width)
- **Return is by branching to the address in `lr`**

```
void func1 (void)
{
        :
        func2();
        :
}
```



Supervisor Call (SVC)

```
SVC{<cond>} <SVC number>
```

- **Causes an SVC exception**
- **The SVC handler can examine the SVC number to decide what operation has been requested**
  - But the core ignores the SVC number
- **By using the SVC mechanism, an operating system can implement a set of privileged operations (system calls) which applications running in user mode can request**
- **Thumb version is unconditional**

Software Interrupt

- ***SWI* instruction**
  - Forces CPU into supervisor mode
  - Usage: SWI #n

31 28	27 24	23 0
Cond	Opcode	Ordinal

- **Maximum 224 calls**
- **Suitable for running privileged code and making OS calls**

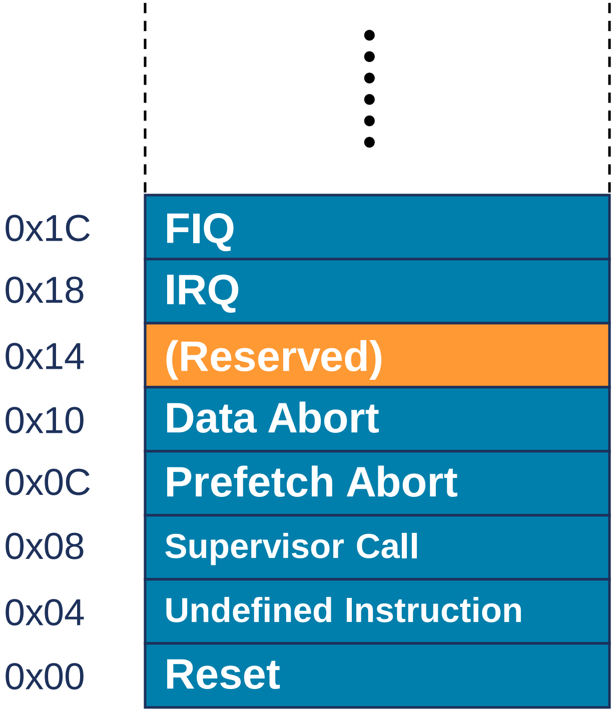
Exception Handling

- **When an exception occurs, the core…**
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state (if appropriate)
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address

- **To return, exception handler needs to…**
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

- **Cores can enter ARM state or Thumb state when taking an exception**
  - Controlled through settings in CP15

- **Note that v7-M and v6-M exception model is different**

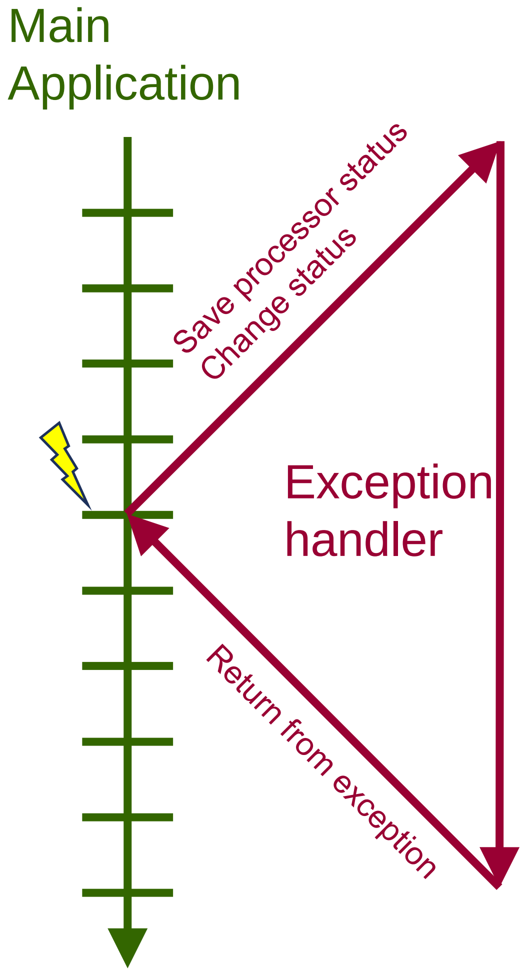


	⋮
0x1C	**FIQ**
0x18	**IRQ**
0x14	**(Reserved)**
0x10	**Data Abort**
0x0C	**Prefetch Abort**
0x08	**Supervisor Call**
0x04	**Undefined Instruction**
0x00	**Reset**

Vector Table

Vector table can also be at **`0xFFFF0000`** on most cores

Exception handling process

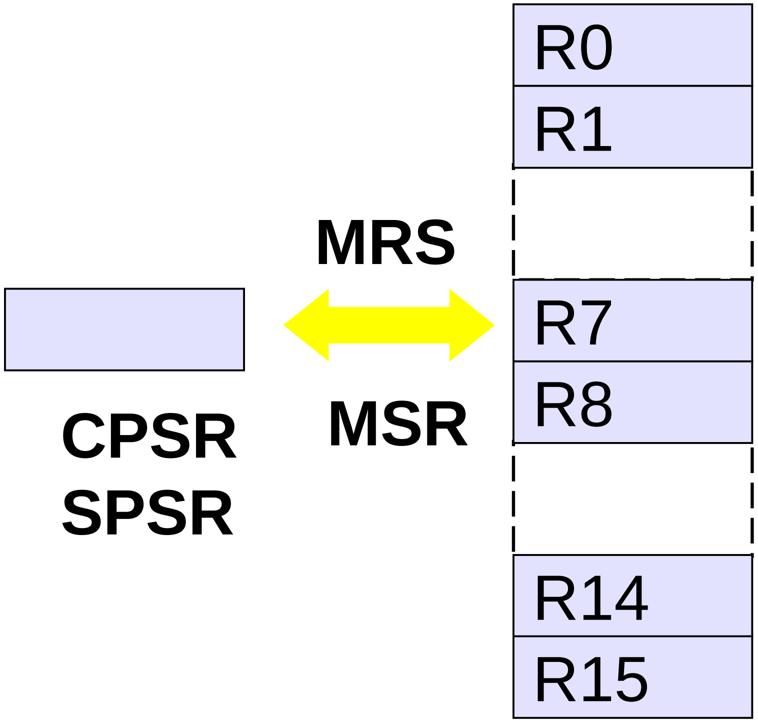


1. **Save processor status**
   - Copies `CPSR` into `SPSR_<mode>`
   - Stores the return address in `LR_<mode>`
   - Adjusts LR based on exception type
2. **Change processor status for exception**
   - Mode field bits
   - ARM or Thumb state
   - Interrupt disable bits (if appropriate)
   - Sets PC to vector address
3. **Execute exception handler**
   - <users code>
4. **Return to main application**
   - Restore `CPSR` from `SPSR_<mode>`
   - Restore PC from `LR_<mode>`

- **1 and 2 performed automatically by the core**
- **3 and 4 responsibility of software**

Modifying the Status Registers

- Only indirectly
- *MSR* moves contents from CPSR/SPSR to selected GPR
- *MRS* moves contents from selected GPR to CPSR/SPSR
- Only in privileged modes



Barrier instructions

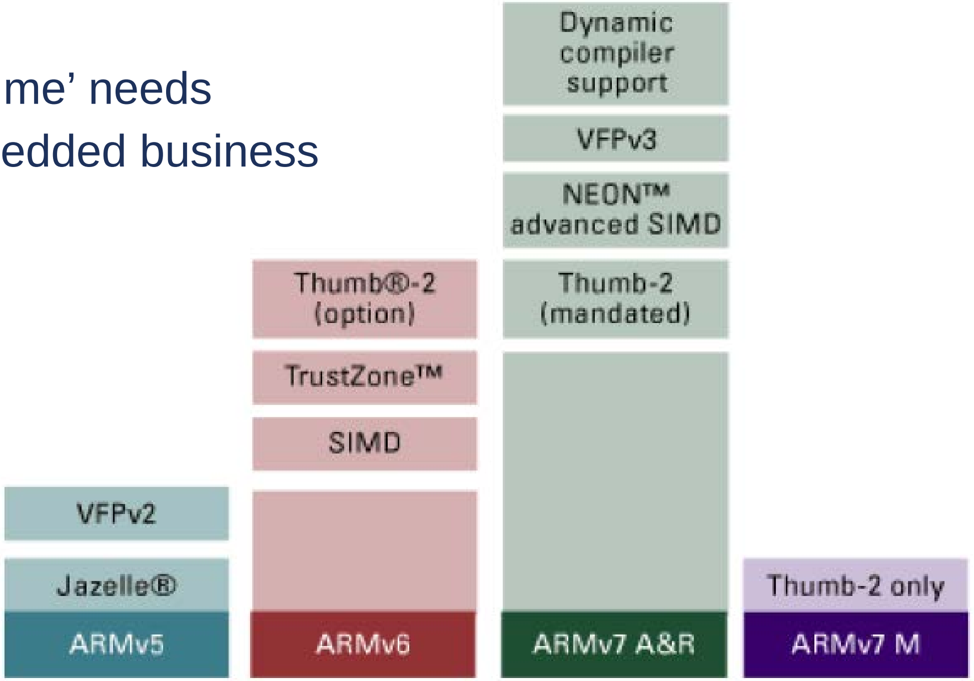
- Useful for multi-core & Self-modifying code

Instruction	Description
DMB	Data memory barrier; ensures that all memory accesses are completed before new memory access is committed
DSB	Data synchronization barrier; ensures that all memory accesses are completed before next instruction is executed
ISB	Instruction synchronization barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Architecture ARMv7-AR profiles

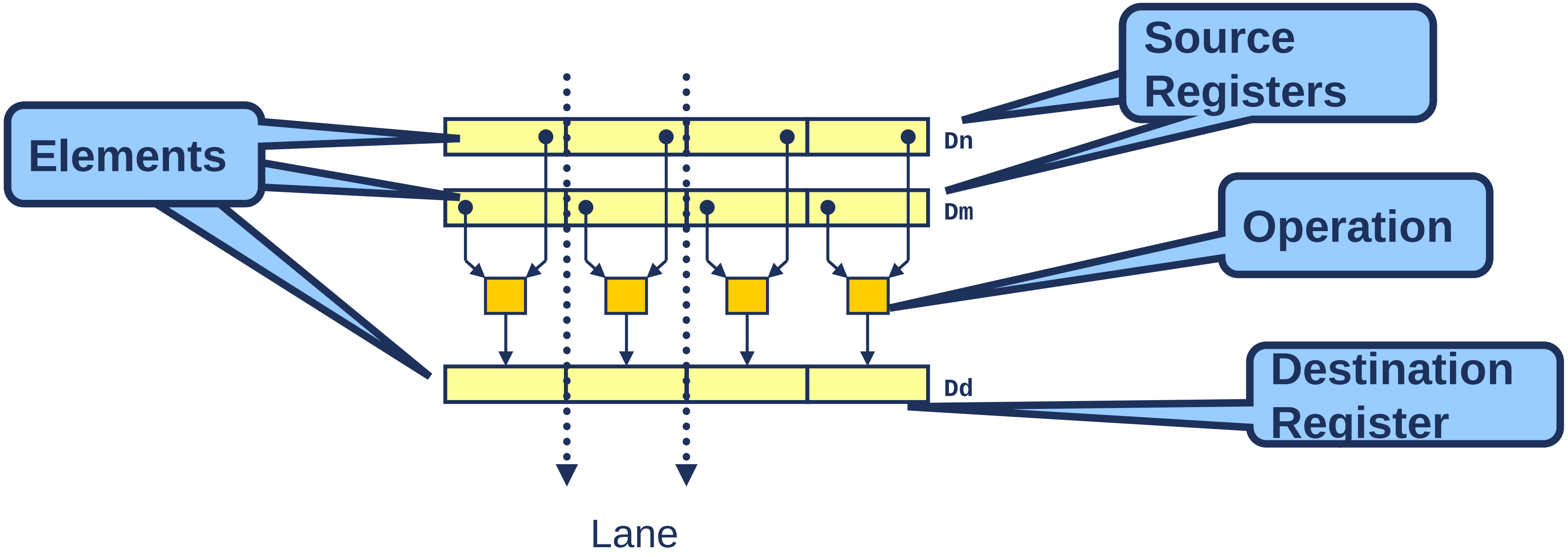
- **Application profile (ARMv7-A)**
  - Memory management support (MMU)
  - Highest performance at low power
  - Influenced by multi-tasking OS system requirements
  - e.g. Cortex-A5, Cortex-A8, Cortex-A9, Cortex-A15

- **Real-time profile (ARMv7-R)**
  - Protected memory (MPU)
  - Low latency and predictability ‘real-time’ needs
  - Evolutionary path for traditional embedded business
  - e.g. Cortex-R4, Cortex-R5



What is NEON?

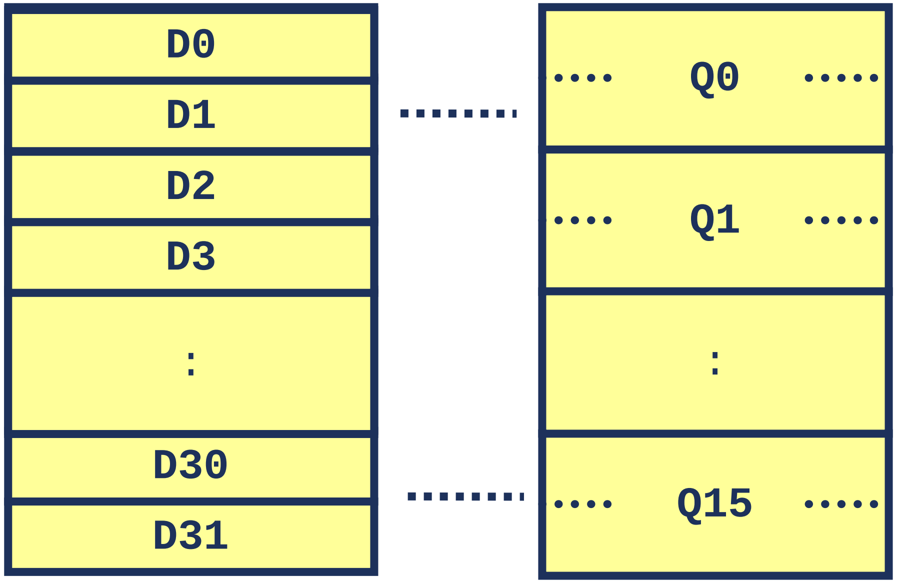
- **NEON is a wide SIMD data processing architecture**
  - Extension of the ARM instruction set (v7-A)
  - 32 x 64-bit wide registers (can also be used as 16 x 128-bit wide registers)
- **NEON instructions perform “Packed SIMD” processing**
  - Registers are considered as **vectors** of **elements** of the same data type
  - Data types available: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single prec. float
  - Instructions usually perform the same operation in all **lanes**



NEON Coprocessor registers

- **NEON has a 256-byte register file**
  - Separate from the core registers (r0-r15)
  - Extension to the VFPv2 register file (VFPv3)

- **Two different views of the NEON registers**
  - 32 x 64-bit registers (D0-D31)
  - 16 x 128-bit registers (Q0-Q15)



- **Enables register trade-offs**
  - Vector length can be variable
  - Different registers available

NEON vectorizing example

- How does the compiler perform vectorization?

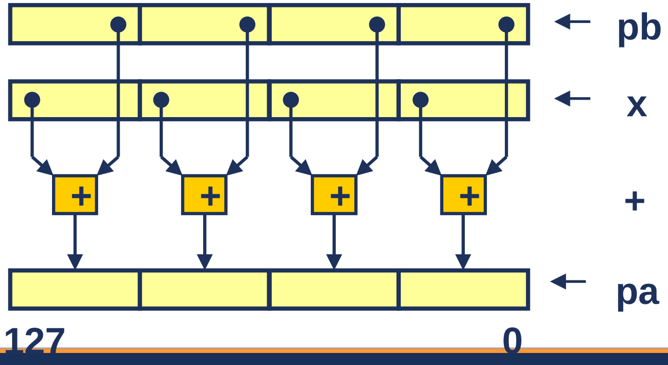
```
void add_int(int * __restrict pa,
             int * __restrict pb,
             unsigned int n, int x)
{
  unsigned int i;
  for(i = 0; i < (n & ~3); i++)
    pa[i] = pb[i] + x;
}
```

1. Analyze each loop:
   - Are pointer accesses safe for vectorization?
   - What data types are being used? How do they map onto NEON vector registers?
   - Number of loop iterations

2. Unroll the loop to the appropriate number of iterations, and perform other transformations like pointerization

```
void add_int(int *pa, int *pb,
             unsigned n, int x)
{
  unsigned int i;
  for (i = ((n & ~3) >> 2); i; i--)
  {
    *(pa + 0) = *(pb + 0) + x;
    *(pa + 1) = *(pb + 1) + x;
    *(pa + 2) = *(pb + 2) + x;
    *(pa + 3) = *(pb + 3) + x;
    pa += 4; pb += 4;
  }
}
```

3. Map each unrolled operation onto a NEON vector lane, and generate corresponding NEON instructions



Advanced processors

Extracting ILP: out-of-order execution and speculation

Limitations of Scalar Pipelines

Upper Bound on Scalar Pipeline Throughput

Limited by IPC=1 "Flynn Bottleneck"

Performance Lost Due to Rigid In-order Pipeline

Unnecessary stalls

Terms

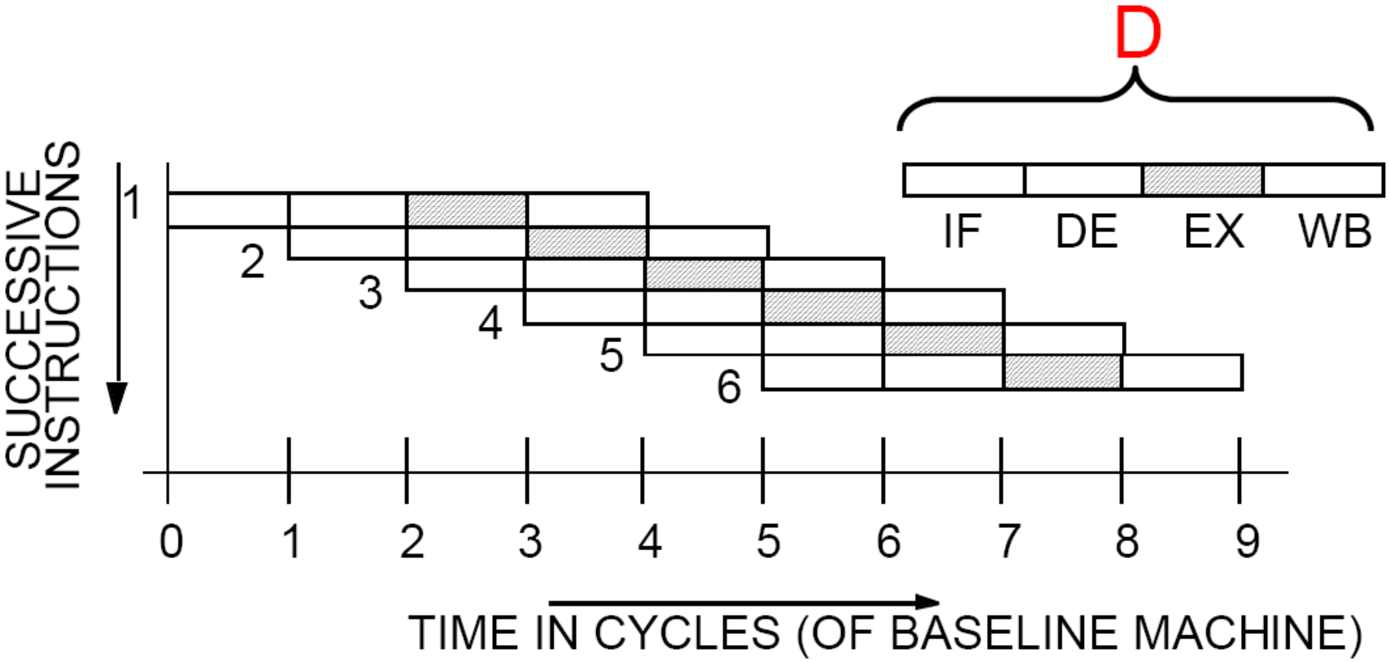
- **Instruction parallelism**
  - Number of instructions being worked on
- **Operation Latency**
  - The time (in cycles) until the result of an instruction is available for use as an operand in a subsequent instruction. For example, if the result of an Add instruction can be used as an operand of an instruction that is issued in the cycle after the Add is issued, we say that the Add has an operation latency of one.
- **Peak IPC**
  - The maximum sustainable number of instructions that can be executed per clock.

# Performance modeling for computer architects, C. M. Krishna

Architectures for Exploiting Instruction-Level Parallelism

Scalar Pipeline (baseline)

- Instruction Parallelism = D
- Operation Latency = 1
- Peak IPC = 1



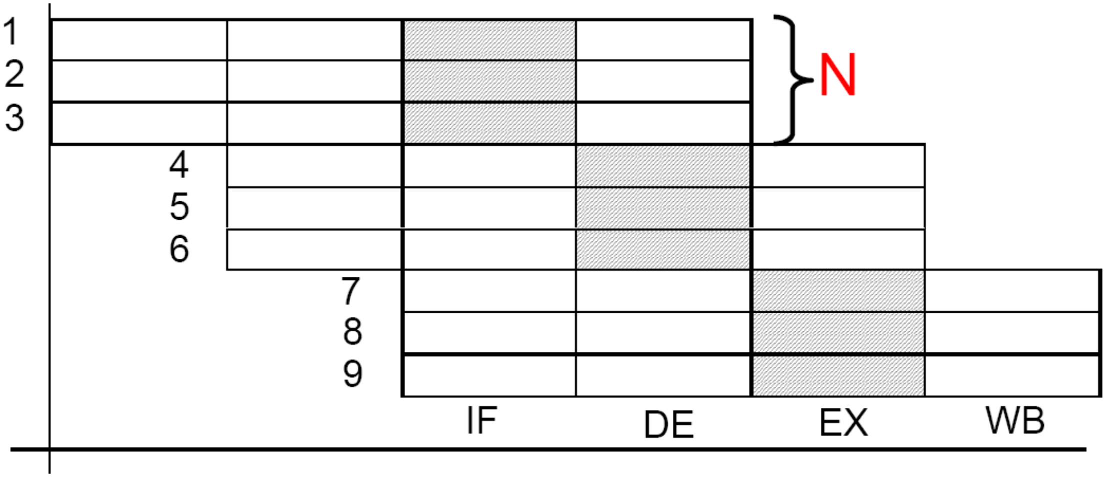
Superscalar Machine

Superscalar (Pipelined) Execution

IP = *DxN*

OL = *1 baseline cycles*

Peak IPC = *N per baseline cycle*



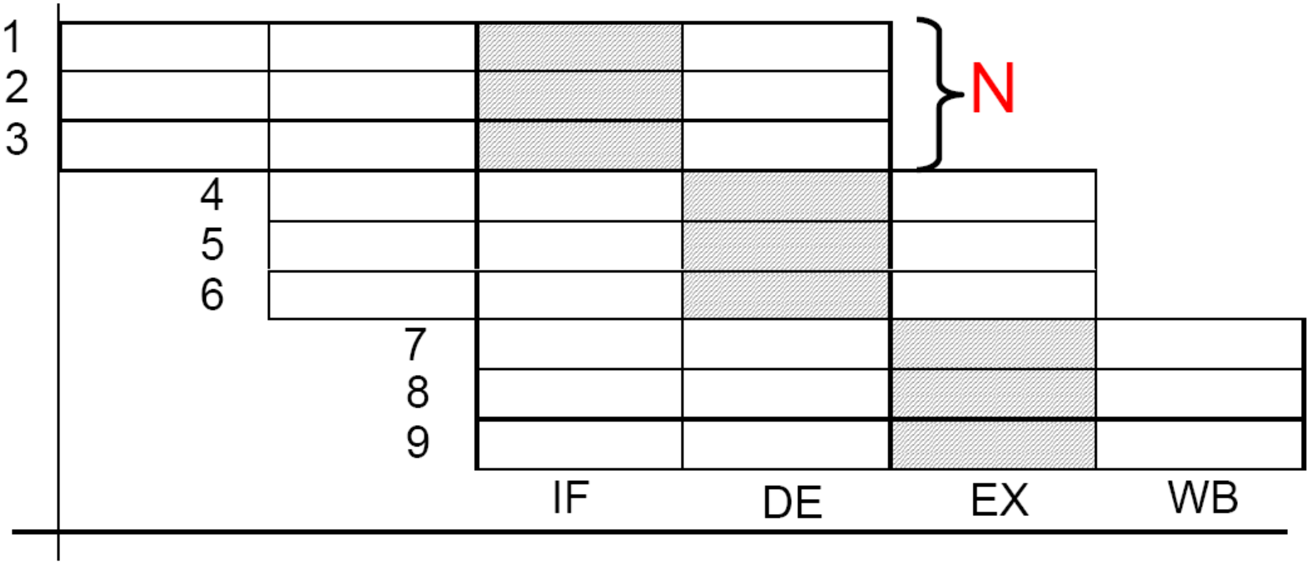
What is the real problem?

CPI of in-order pipelines degrades very sharply if the machine parallelism is increased beyond a certain point.

i.e., when NxM approaches average distance between dependent instructions

Forwarding is no longer effective

Pipeline may never be full due to frequent dependency stalls!



Missed Speedup in In-Order Pipelines

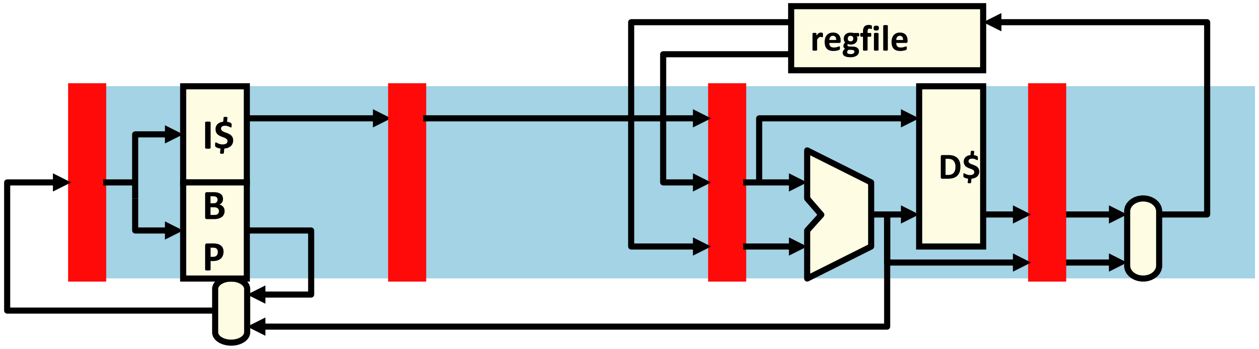
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`addf f0,f1,f2`	F	D	E+	E+	E+	W										
`mulf f2,f3,f2`		F	D	d*	d*	E*	E*	E*	E*	E*	W					
`subf f0,f1,f4`			F	p*	p*	D	E+	E+	E+	W						

What's happening in cycle 4?

- **`mulf`** stalls due to **RAW hazard**
  - OK, this is a fundamental problem
- **`subf`** stalls due to **pipeline hazard**
  - Why? **`subf`** can't proceed into D because **`mulf`** is there
  - That is the only reason, and it isn't a fundamental one

Why can't **`subf`** go into D in cycle 4 and E+ in cycle 5?

The Problem With In-Order Pipelines



- **In-order pipeline**
  - Often written as F,D,X,W (multi-cycle X includes M)
  - Variable latency
    - 1-cycle integer (including mem)
    - 3-cycle pipelined FP
  - **Structural hazard**: 1 insn register (latch) per stage
    - 1 instruction per stage per cycle (unless pipeline is replicated)
    - Younger instr. can’t “pass” older instr. without “clobbering” it
- **Out-of-order pipeline**
  - Implement “passing” functionality by removing structural hazard

ILP: Instruction-Level Parallelism

ILP is a measure of the amount of inter-dependencies between instructions

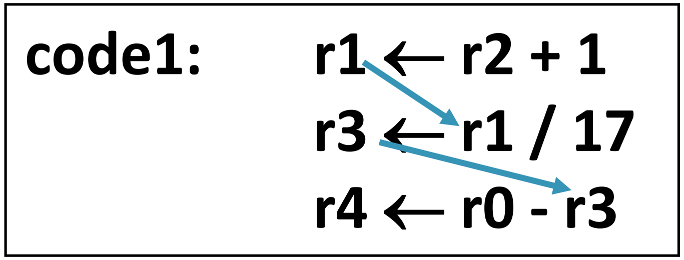
Average ILP =no. instruction / no. cyc required

code1: ILP = 1

i.e. must execute serially

code2: ILP = 3

i.e. can execute at the same time



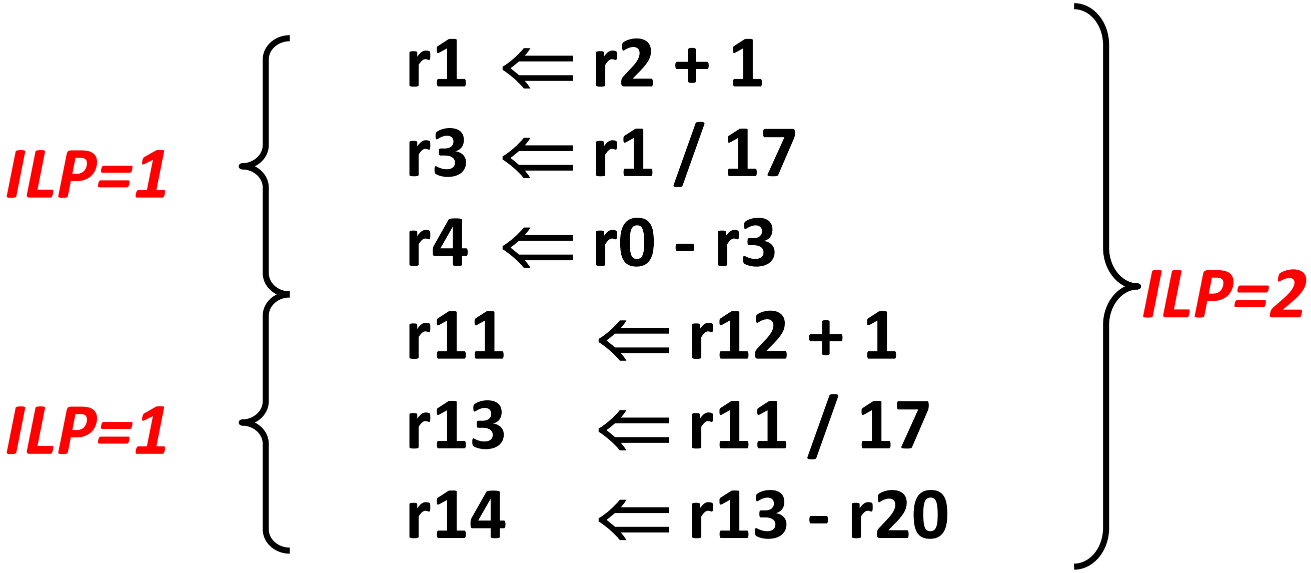
code1:

r1 ← r2 + 1
r3 ← r1 / 17
r4 ← r0 - r3

code2:

r1 ← r2 + 1
r3 ← r9 / 17
r4 ← r0 - r10

Scope of ILP Analysis



ILP=1

r1 ⇐ r2 + 1
r3 ⇐ r1 / 17
r4 ⇐ r0 - r3

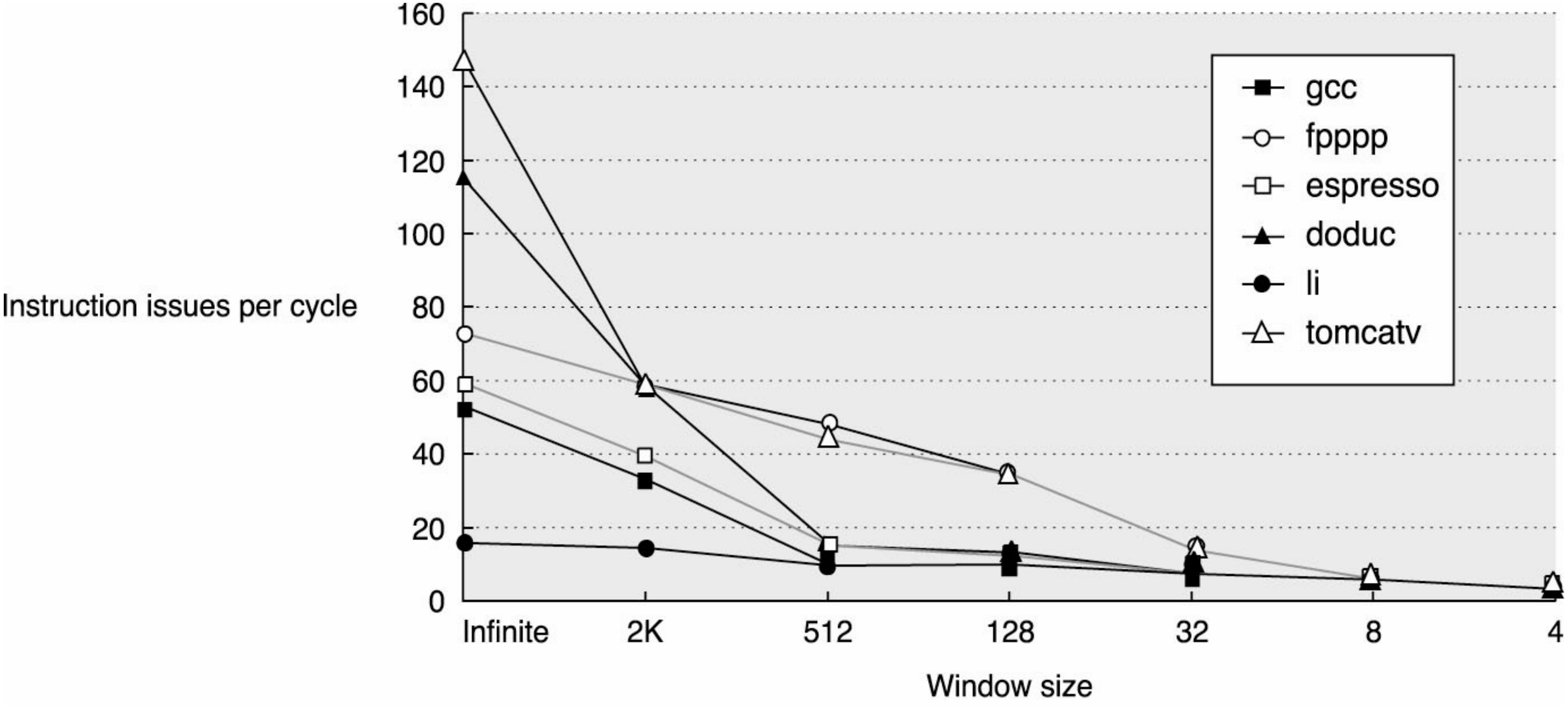
ILP=1

r11 ⇐ r12 + 1
r13 ⇐ r11 / 17
r14 ⇐ r13 - r20

ILP=2

Out-of-order execution exposes more ILP

How Large Must the "Window" Be?



© 2003 Elsevier Science (USA). All rights reserved.

Dynamic Scheduling – OoO Execution

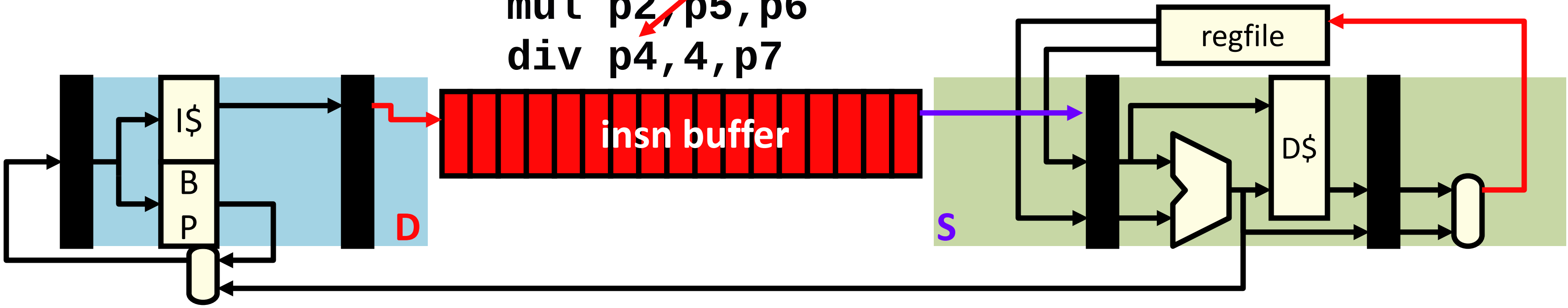
- Dynamic scheduling
  - Totally in the hardware
  - Also called “out-of-order execution” (OoO)
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline on branch misprediction
- Rename to avoid false dependencies (WAW and WAR)
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies more tricky (much more later)
- Commit instructions in order
  - Any strange happens before commit, just flush the pipeline
- Current machines: 100+ instruction scheduling window

Motivation for Dynamic Scheduling

- **Dynamic scheduling (out-of-order execution)**
  - – Execute instructions in non-sequential order…
    - + Reduce RAW stalls
    - + Increase pipeline and functional unit (FU) utilization
      - – Original motivation was to increase FP unit utilization
    - + Expose more opportunities for parallel issue (ILP)
      - – Not in-order → can be in parallel
  - – …but make it appear like sequential execution
    - Important
      - – But difficult

Dynamic Scheduling: The Big Picture

```
add p2,p3,p4
sub p2,p4,p5
mul p2,p5,p6
div p4,4,p7
```



Ready Table

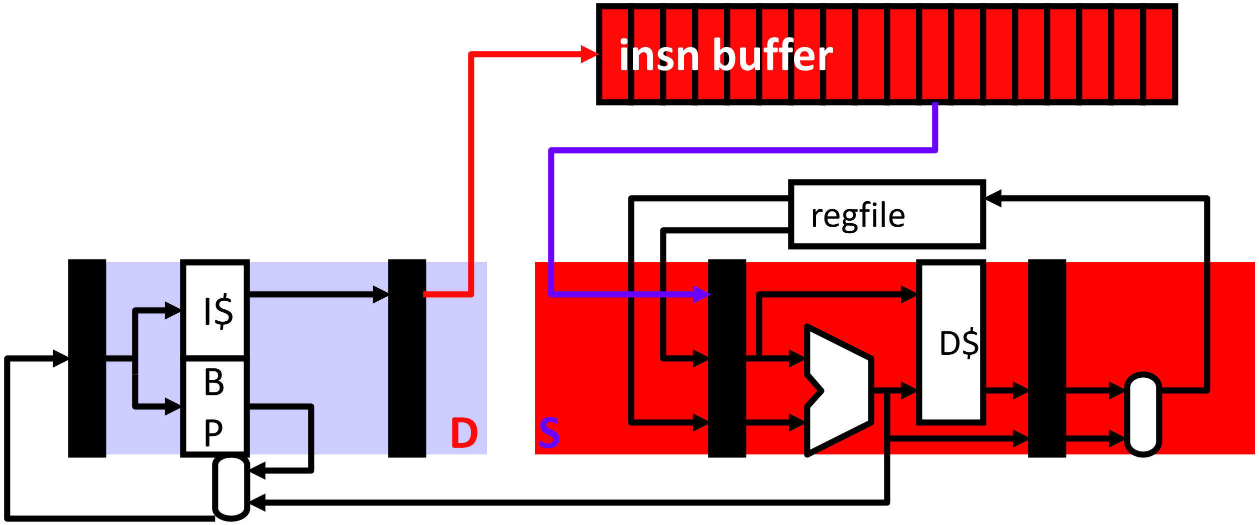
t

P2	P3	P4	P5	P6	P7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

```
add p2,p3,p4
sub p2,p4,p5  and  div p4,4,p7
mul p2,p5,p6
```

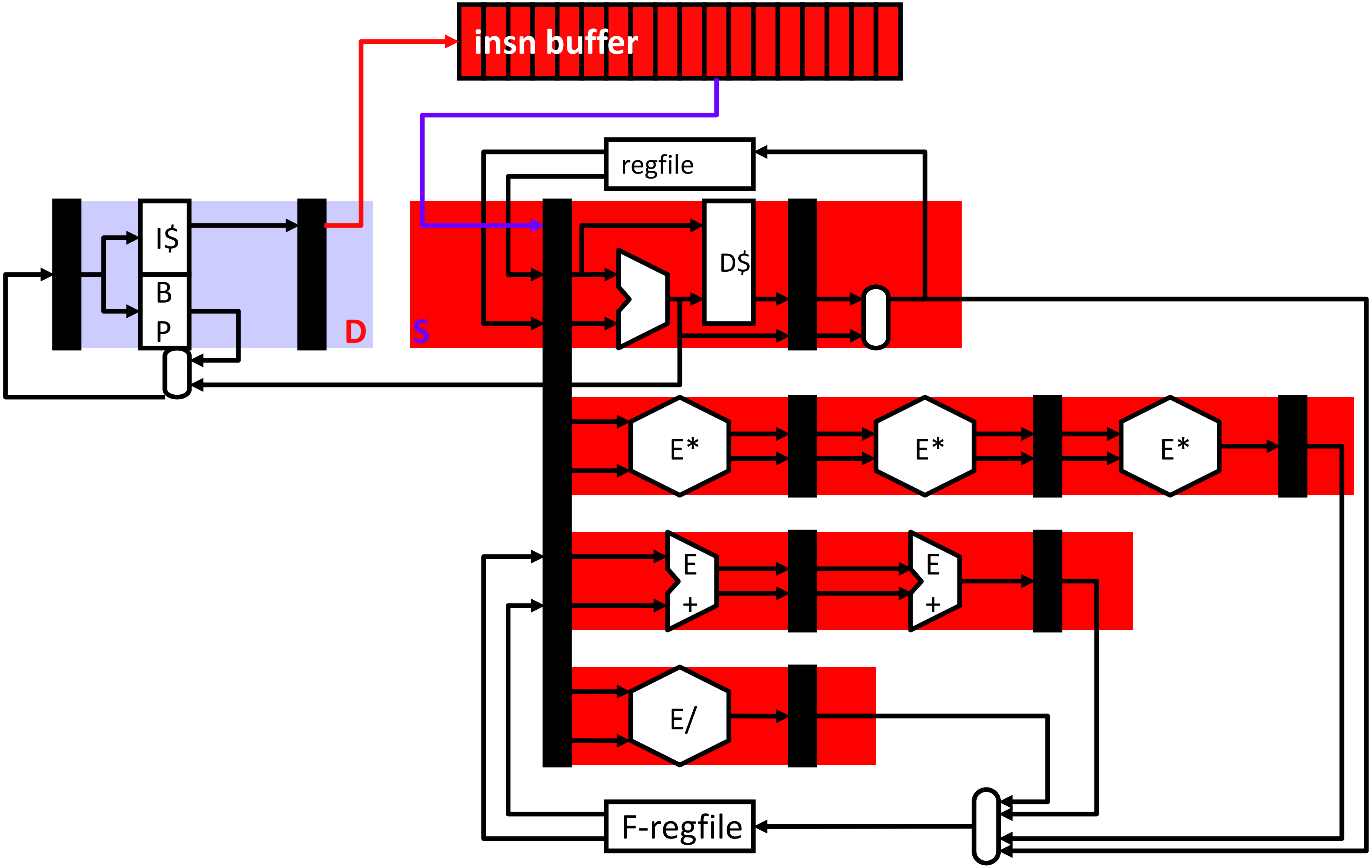
- Instructions fetch/decoded/renamed into *Instruction Buffer*
  - Also called “instruction window” or “instruction scheduler”
- Instructions (conceptually) check ready bits every cycle
  - Execute when ready

Anatomy of OoO: Dispatch and Issue



- **Dispatch (D)**: first part of decode
  - Allocate slot in insn buffer
    - New kind of structural hazard (insn buffer is full)
  - In order: **stall** back-propagates to younger insns
- **Issue (S)**: second part of decode
  - Send insns from insn buffer to execution units
  - Out-of-order: **wait** doesn’t back-propagate to younger insns

Dispatch and Issue with Floating-Point



Key OoO Design Feature: Issue Policy and Issue Logic

- Issue
  - If multiple instructions are ready, which one to choose? **Issue policy**
    - Oldest first? Safe
    - Longest latency first? May yield better performance
  - **Select logic**: implements issue policy
    - Most projects use random.

Going Forward: What's Next

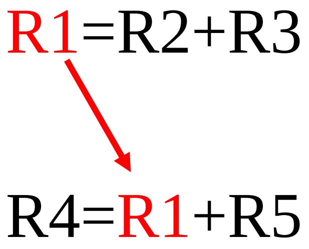
- We'll build this up in steps
  - Register renaming to eliminate "false" dependencies
  - "Tomasulo's algorithm" to implement OoO execution
  - Handling precise state and speculation
  - Handling memory dependencies

Dependency vs. Hazard

- A dependency exists *independent* of the hardware.
  - So if Inst #1's result is needed for Inst #1000 there is a dependency
  - It is only a *hazard* if the hardware has to deal with it.
    - So in our pipelined machine we only worried if there wasn't a "buffer" of two instructions between the dependent instructions.

True Data dependencies

- True data dependency
  - RAW – Read after Write



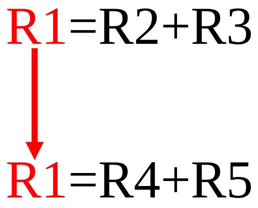
R1=R2+R3

R4=R1+R5

- True dependencies prevent reordering
  - (Mostly) unavoidable

False Data Dependencies

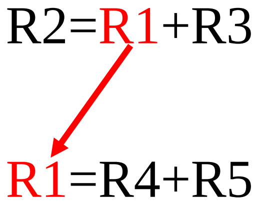
- False or Name dependencies
  - WAW – Write after Write



R1=R2+R3

R1=R4+R5

  - WAR – Write after Read



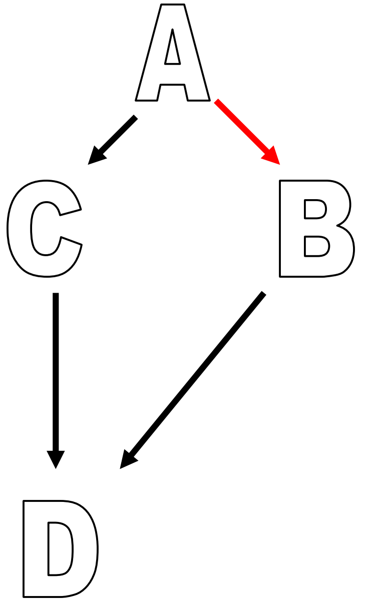
R2=R1+R3

R1=R4+R5

- False dependencies prevent reordering
  - Can they be eliminated? (Yes, with renaming!)

Data Dependency Graph: Simple example

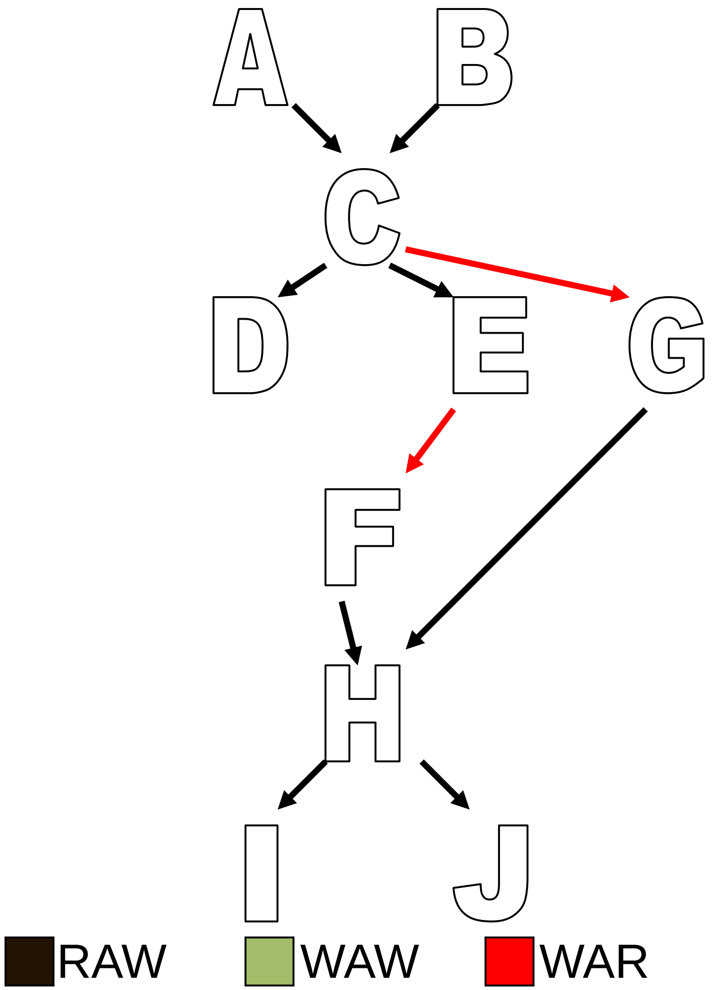
```
R1=MEM[R2+0]   // A
R2=R2+4        // B
R3=R1+R4       // C
MEM[R2+0]=R3   // D
```



⬛RAW 🟩WAW 🟥WAR

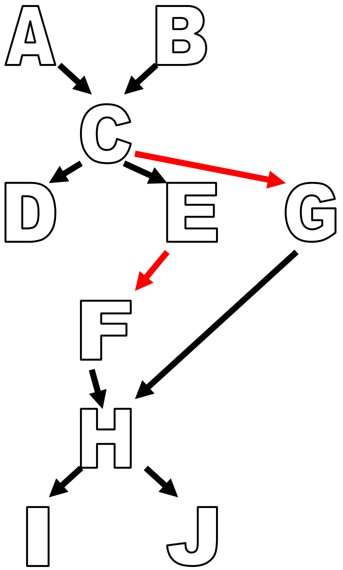
Data Dependency Graph: More complex example

```
R1=MEM[R3+4]     // A
R2=MEM[R3+8]     // B
R1=R1*R2         // C
MEM[R3+4]=R1     // D
MEM[R3+8]=R1     // E
R1=MEM[R3+12]    // F
R2=MEM[R3+16]    // G
R1=R1*R2         // H
MEM[R3+12]=R1    // I
MEM[R3+16]=R1    // J
```



Eliminating False Dependencies

```
R1=MEM[R3+4]    // A
R2=MEM[R3+8]    // B
R1=R1*R2        // C
MEM[R3+4]=R1    // D
MEM[R3+8]=R1    // E
R1=MEM[R3+12]   // F
R2=MEM[R3+16]   // G
R1=R1*R2        // H
MEM[R3+12]=R1   // I
MEM[R3+16]=R1   // J
```



- Well, logically there is no reason for F-J to be dependent on A-E. So…..
    - ABFG
    - CH
    - DEIJ
  - Should be possible.
- But that would cause either C or H to have the wrong reg inputs
- How do we fix this?
  - Remember, the dependency is really on the *name* of the register
  - So… change the register names!

Register Renaming Concept

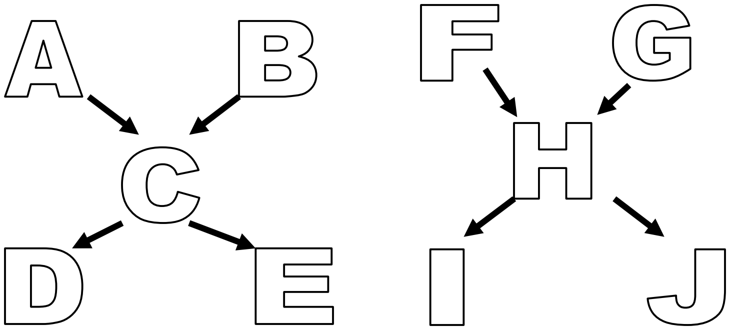
- The register names are arbitrary
- The register name only needs to be consistent between writes.

R1= …..
…. = R1 ….
….= … R1
R1= ….

The value in R1 is "alive" from when the value is written until the last read of that value.

So after renaming, what happens to the dependencies?

```
P1=MEM[R3+4]    //A
P2=MEM[R3+8]    //B
P3=P1*P2        //C
MEM[R3+4]=P3    //D
MEM[R3+8]=P3    //E
P4=MEM[R3+12]   //F
P5=MEM[R3+16]   //G
P6=P4*P5        //H
MEM[R3+12]=P6   //I
MEM[R3+16]=P6   //J
```



■ RAW 🟩 WAW 🟥 WAR

Register Renaming Approach

- Every time an architected register is written we assign it to a physical register
  - Until the architected register is written again, we continue to translate it to the physical register number
  - Leaves **RAW** dependencies intact
- It is really simple, let's look at an example:
  - Names: **`r1,r2,r3`**
  - Locations: **`p1,p2,p3,p4,p5,p6,p7`**
  - Original mapping: **`r1`**→**`p1`**, **`r2`**→**`p2`**, **`r3`**→**`p3`**, **`p4`**–**`p7`** are "free"

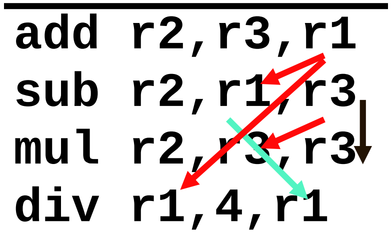
MapTable

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6

FreeList

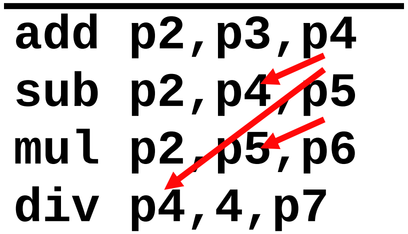
p4,p5,p6,p7
p5,p6,p7
p6,p7
p7

Orig. insns



```
add r2,r3,r1
sub r2,r1,r3
mul r2,r3,r3
div r1,4,r1
```

Renamed insns



```
add p2,p3,p4
sub p2,p4,p5
mul p2,p5,p6
div p4,4,p7
```

Register Renaming

- Register renaming is provided by reservation stations (RS)
    - Contains:
        - The instruction
        - Buffered operand values (when available)
        - Reservation station number of instruction providing the operand values
    - RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
    - Pending instructions designate the RS to which they will send their output
        - Result values broadcast on a result bus, called the common data bus (CDB)
    - Only the last output updates the register file
    - As instructions are issued, the register specifiers are renamed with the reservation station
    - May be more reservation stations than registers

Register vs. Memory Dependence

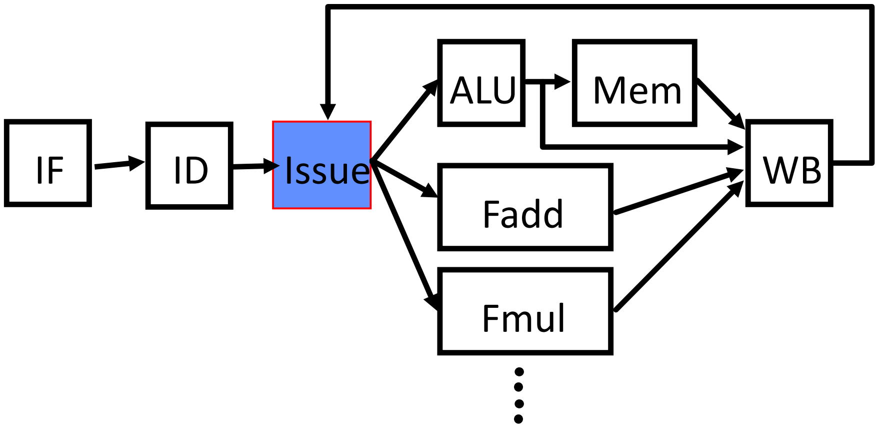
Data hazards due to register operands can be determined at the decode stage, but data hazards due to memory operands can be determined only after computing the effective address

Store: **`M[r1 + disp1] → r2`**

Load: **`r3 ← M[r4 + disp2]`**

Does **`(r1 + disp1) = (r4 + disp2)`** ?

Register Renaming



- Decode does register renaming and adds instructions to the issue-stage instruction **reorder buffer (ROB)**

  →renaming makes WAR or WAW hazards impossible

- Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.

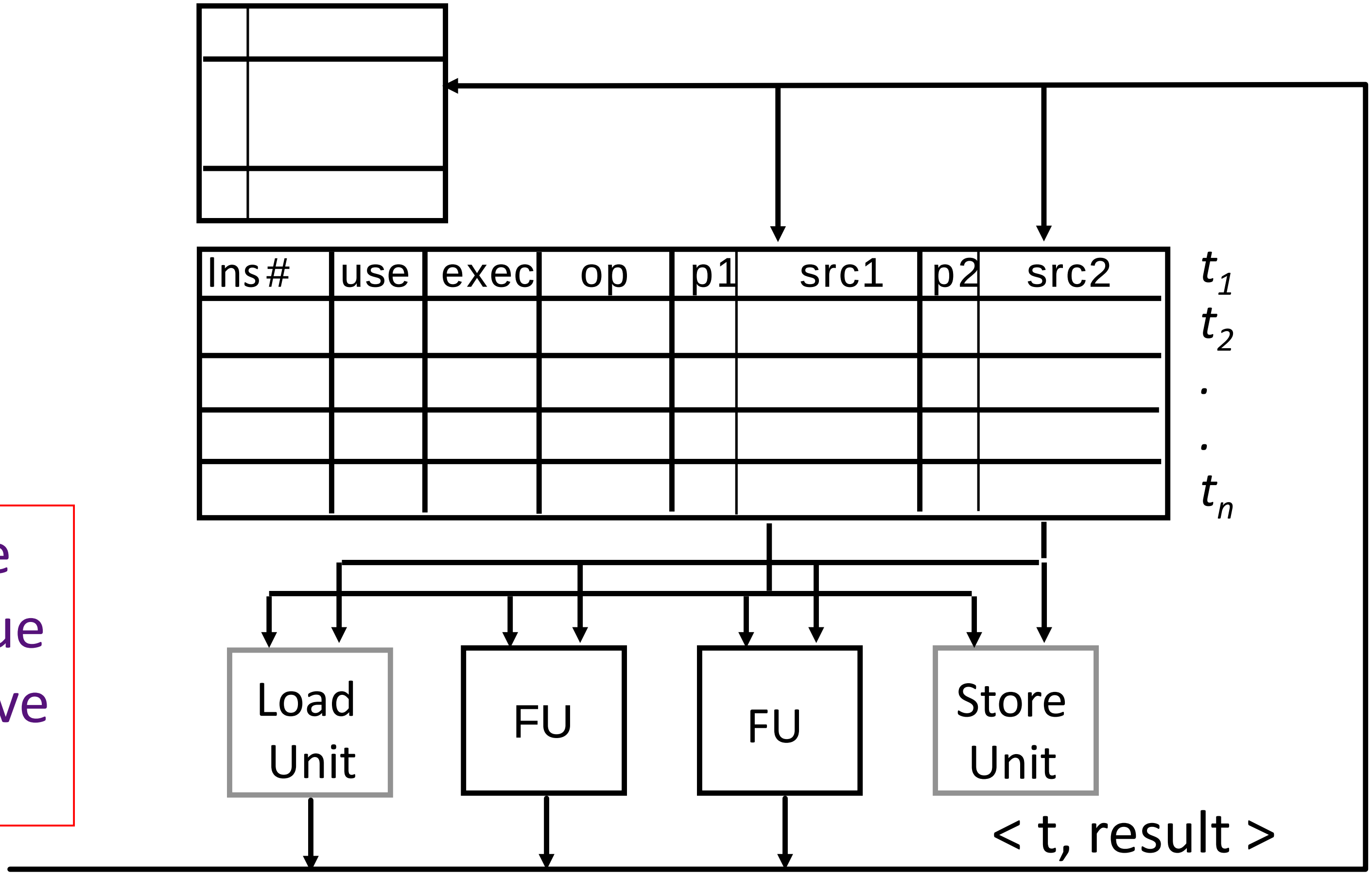
  → Out-of-order or dataflow execution

Renaming Structures

Renaming table & regfile

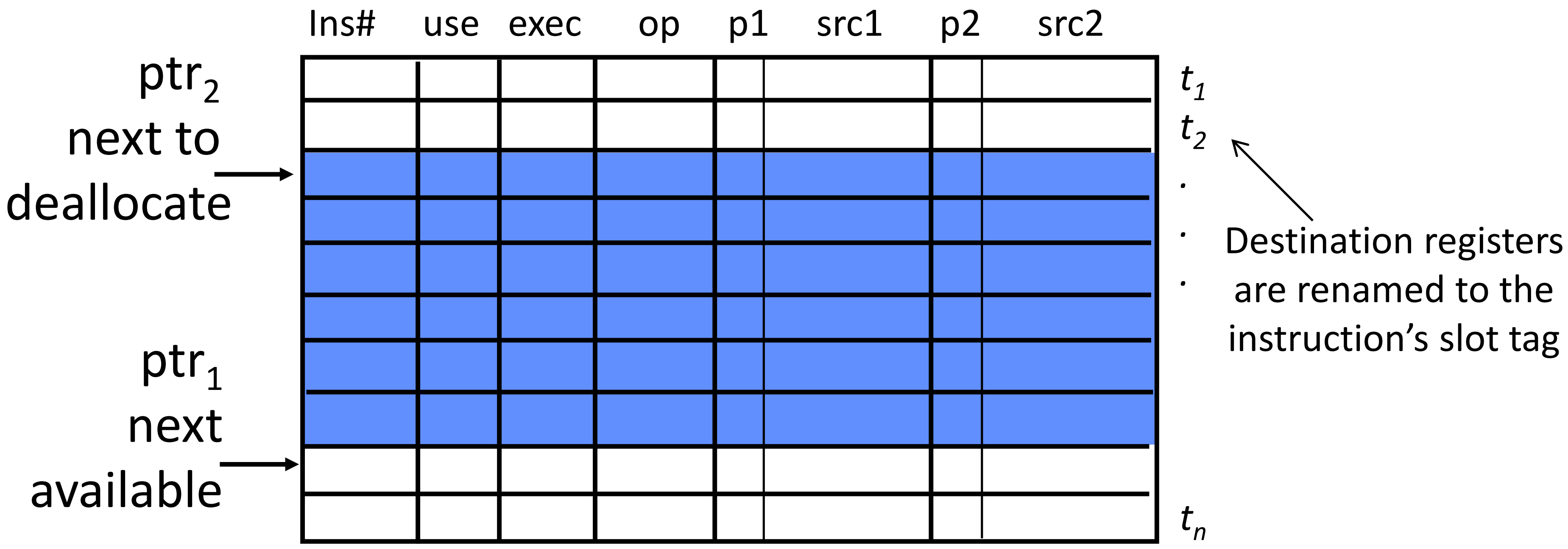
Reorder buffer

Replacing the tag by its value is an expensive operation



- Instruction template (i.e., tag t) is allocated by the Decode stage, which also associates tag with register in regfile
- When an instruction completes, its tag is deallocated

Reorder Buffer Management


Ins#
use
exec
op
p1
src1
p2
src2
ptr_2 next to deallocate
ptr_1 next available
t_1
t_2
.
.
.
t_n
Destination registers are renamed to the instruction's slot tag

ROB managed circularly

- "exec" bit is set when instruction begins execution
- When an instruction completes its "use" bit is marked free
- ptr_2 is incremented only if the "use" bit is marked free

Instruction slot is candidate for execution when:

- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available (p1 and p2 are set)

Renaming & Out-of-order Issue

An example

v1

Renaming table

	p	data
f1		
f2		t1 v1
f3		
f4		t2 t5
f5		
f6		t3
f7		
f8		t4 v4

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
1	0 1	0 1	LD					t_1
2	1 0	0 1	LD					t_2
3	1	0	MUL	1 0	t2 v2	1	t1 v1	t_3
4	1 0	0 1	SUB	1	v1	1	v1	t_4
5	1	0	DIV	1	v1	0 1	t4 v4	t_5
								.
								.

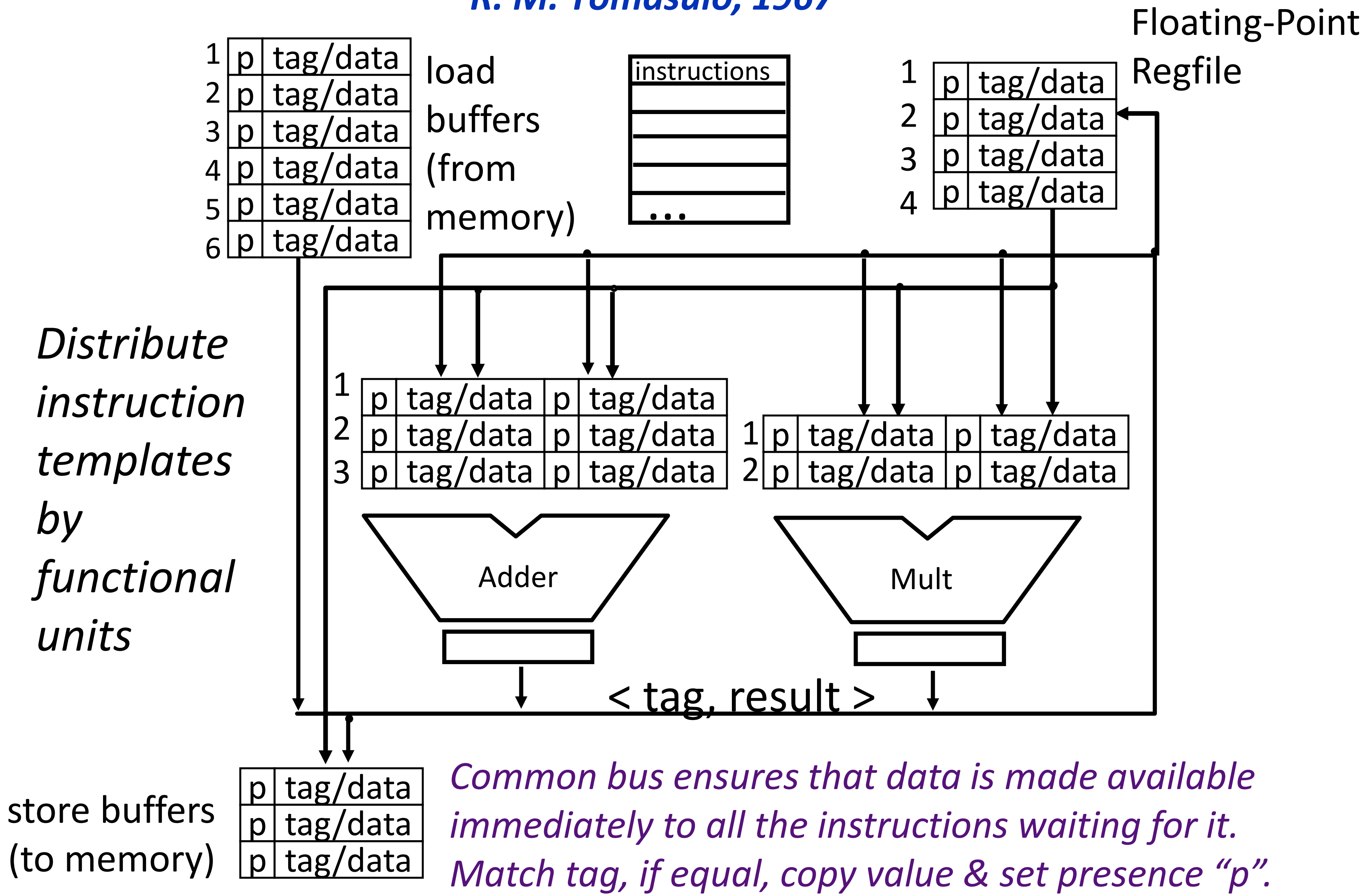
data / t_i

1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- *When are tags in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

IBM 360/91 Floating-Point Unit

R. M. Tomasulo, 1967



1 p tag/data
2 p tag/data
3 p tag/data
4 p tag/data
5 p tag/data
6 p tag/data

load buffers (from memory)

instructions
...

Floating-Point Regfile

1 p tag/data
2 p tag/data
3 p tag/data
4 p tag/data

1 p tag/data p tag/data
2 p tag/data p tag/data
3 p tag/data p tag/data

1 p tag/data p tag/data
2 p tag/data p tag/data

store buffers (to memory)

p tag/data
p tag/data
p tag/data

Distribute instruction templates by functional units

Common bus ensures that data is made available immediately to all the instructions waiting for it. Match tag, if equal, copy value & set presence “p”.

Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but did not show up in the subsequent models until mid-Nineties.

Why ?

Reasons

1. Effective on a very small class of programs
2. Memory latency a much bigger problem
3. Exceptions not precise!

One more problem needed to be solved

Control transfers

Precise Interrupts

It must appear as if an interrupt is taken between two instructions (say I_i and I_{i+1})

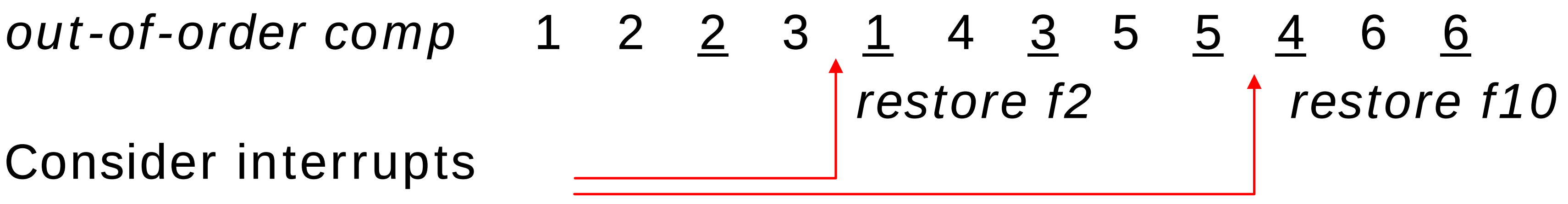
- the effect of all instructions up to and including I_i is totally complete
- no effect of any instruction after I_i has taken place

The interrupt handler either aborts the program or restarts it at I_{i+1} .

Effect on Interrupts

Out-of-order Completion

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2



out-of-order comp 1 2 $\underline{2}$ 3 $\underline{1}$ 4 $\underline{3}$ 5 $\underline{5}$ $\underline{4}$ 6 $\underline{6}$

restore f2

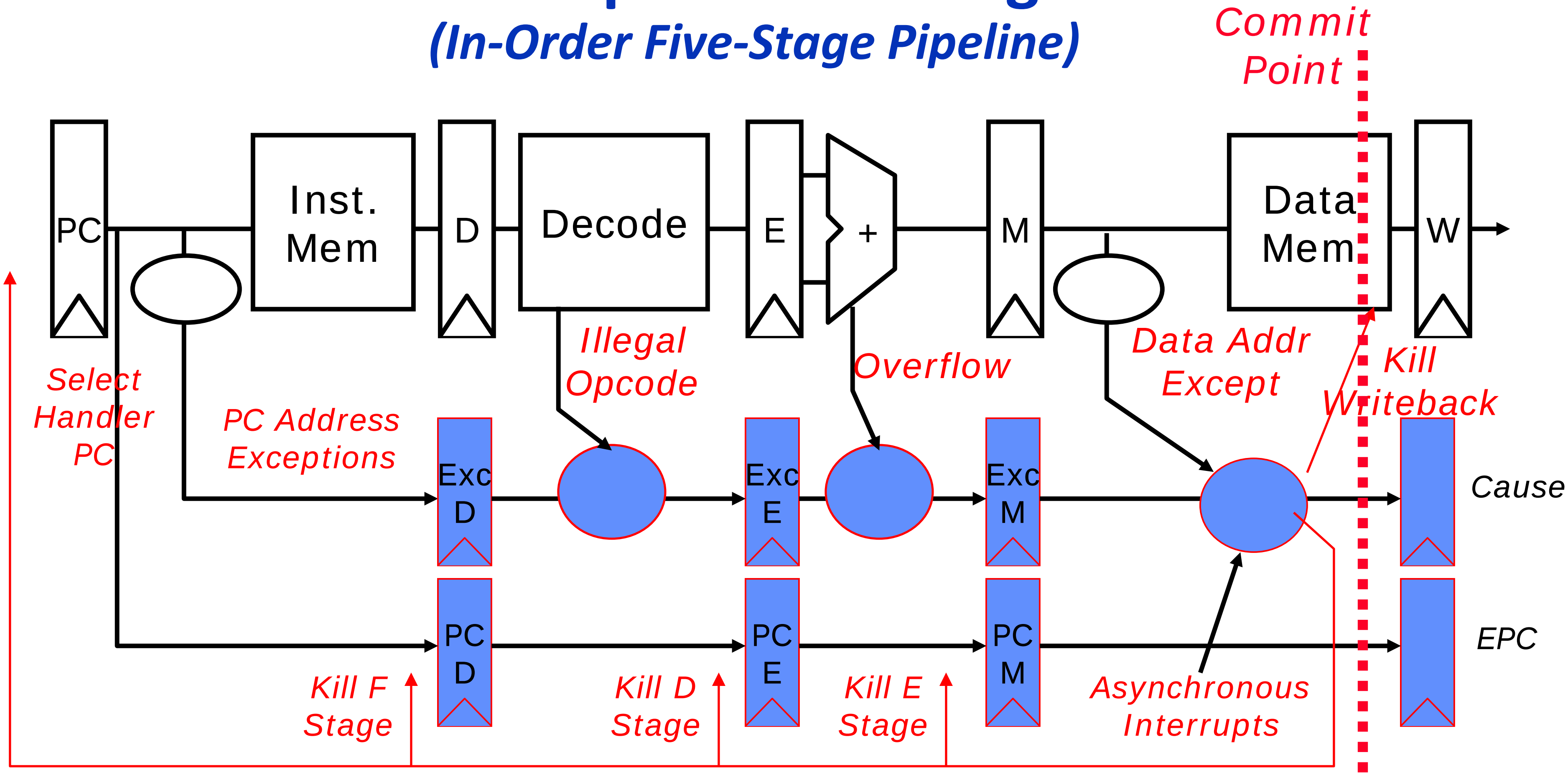
restore f10

Consider interrupts

Precise interrupts are difficult to implement at high speed

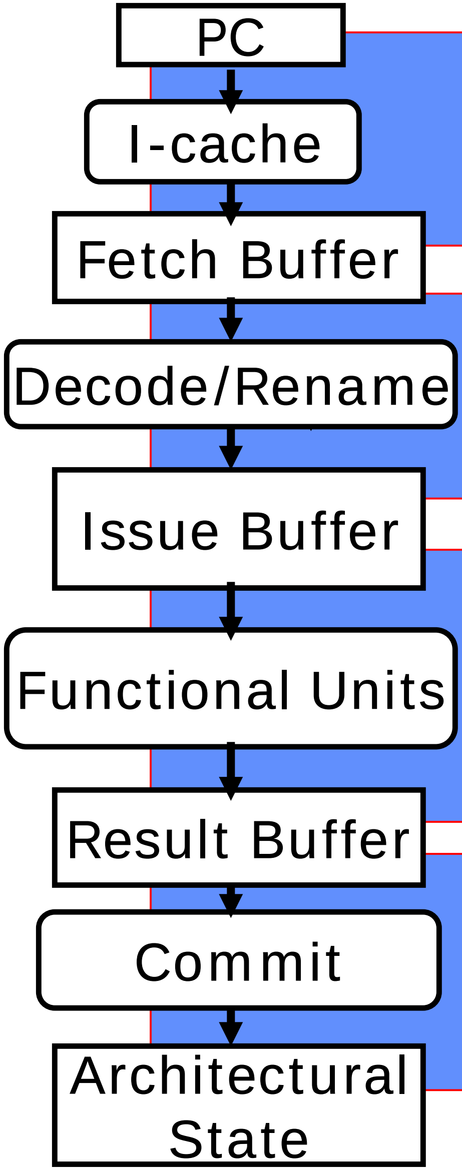
- want to start execution of later instructions before exception checks finished on earlier instructions

Exception Handling
(In-Order Five-Stage Pipeline)



- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Phases of Instruction Execution



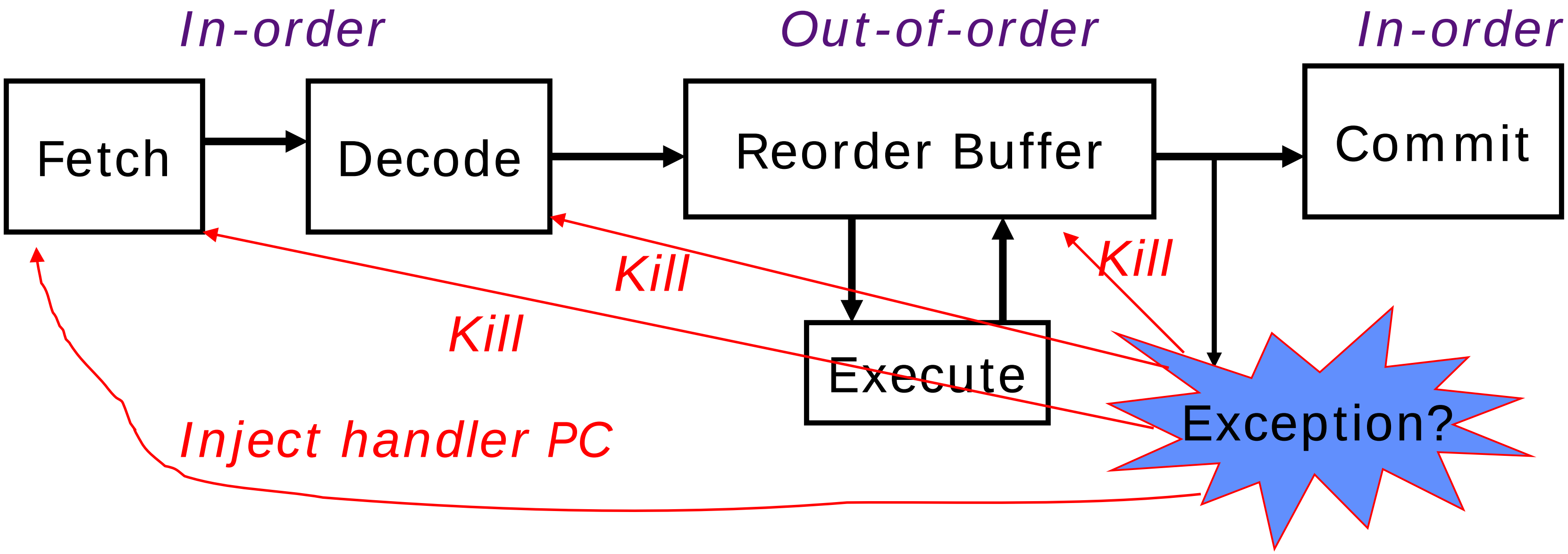
Fetch: Instruction bits retrieved from cache.

Decode: Instructions dispatched to appropriate issue-stage buffer

Execute: Instructions and operands issued to execution units. When execution completes, all results and exception flags are available.

Commit: Instruction irrevocably updates architectural state (aka “graduation”).

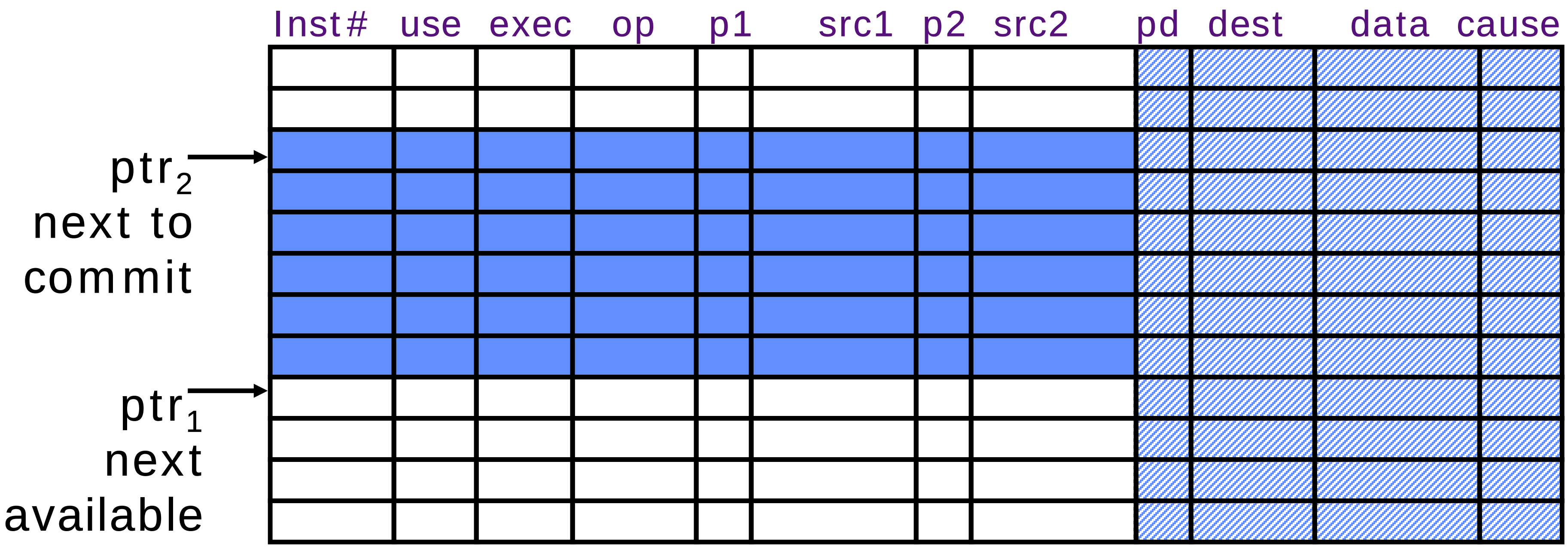
In-Order Commit for Precise Exceptions



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order

Temporary storage needed to hold results before commit (shadow registers and store buffers)

Extensions for Precise Exceptions

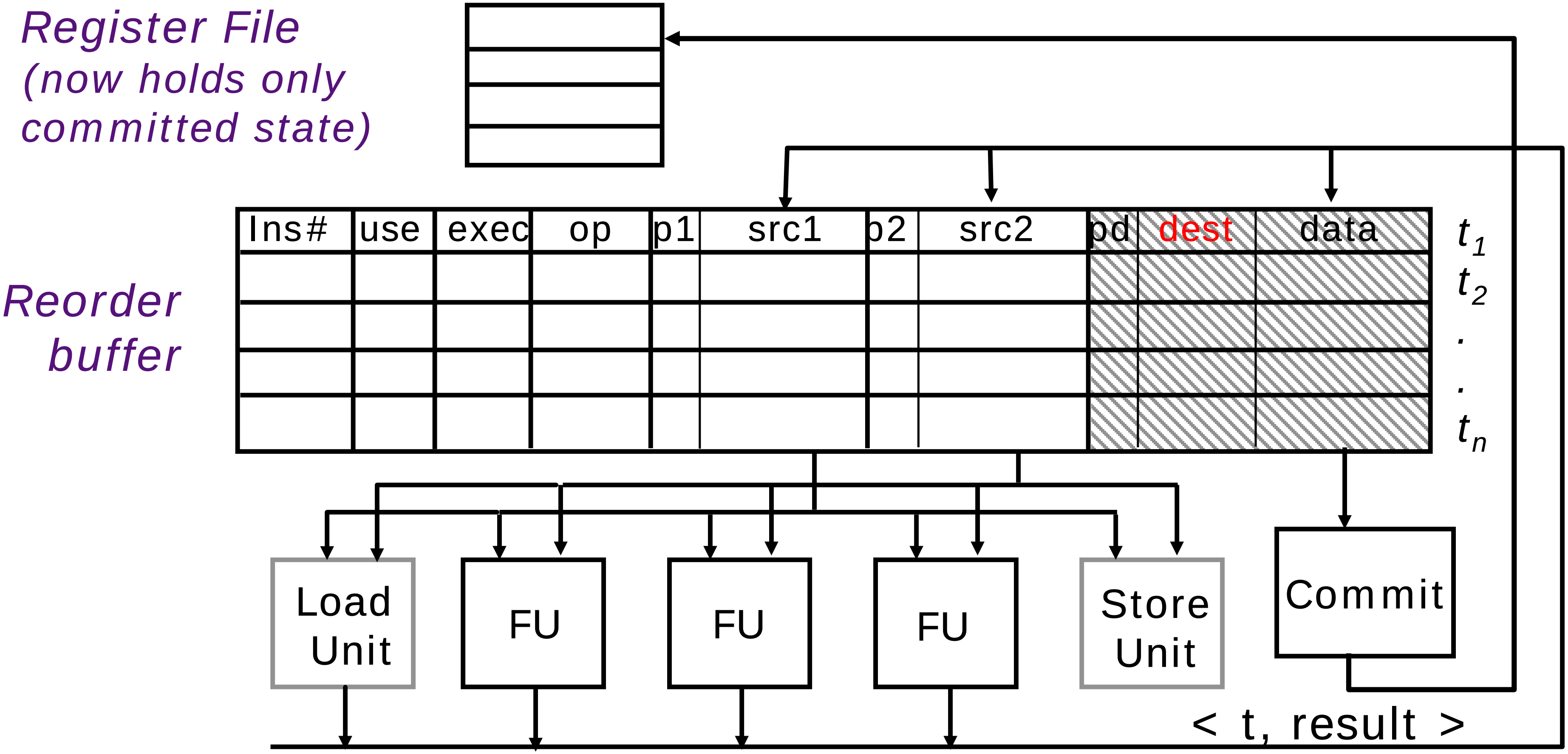


Reorder buffer

- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting $ptr_1 = ptr_2$

  (stores must wait for commit before updating memory)

Rollback and Renaming



Register File
(now holds only committed state)

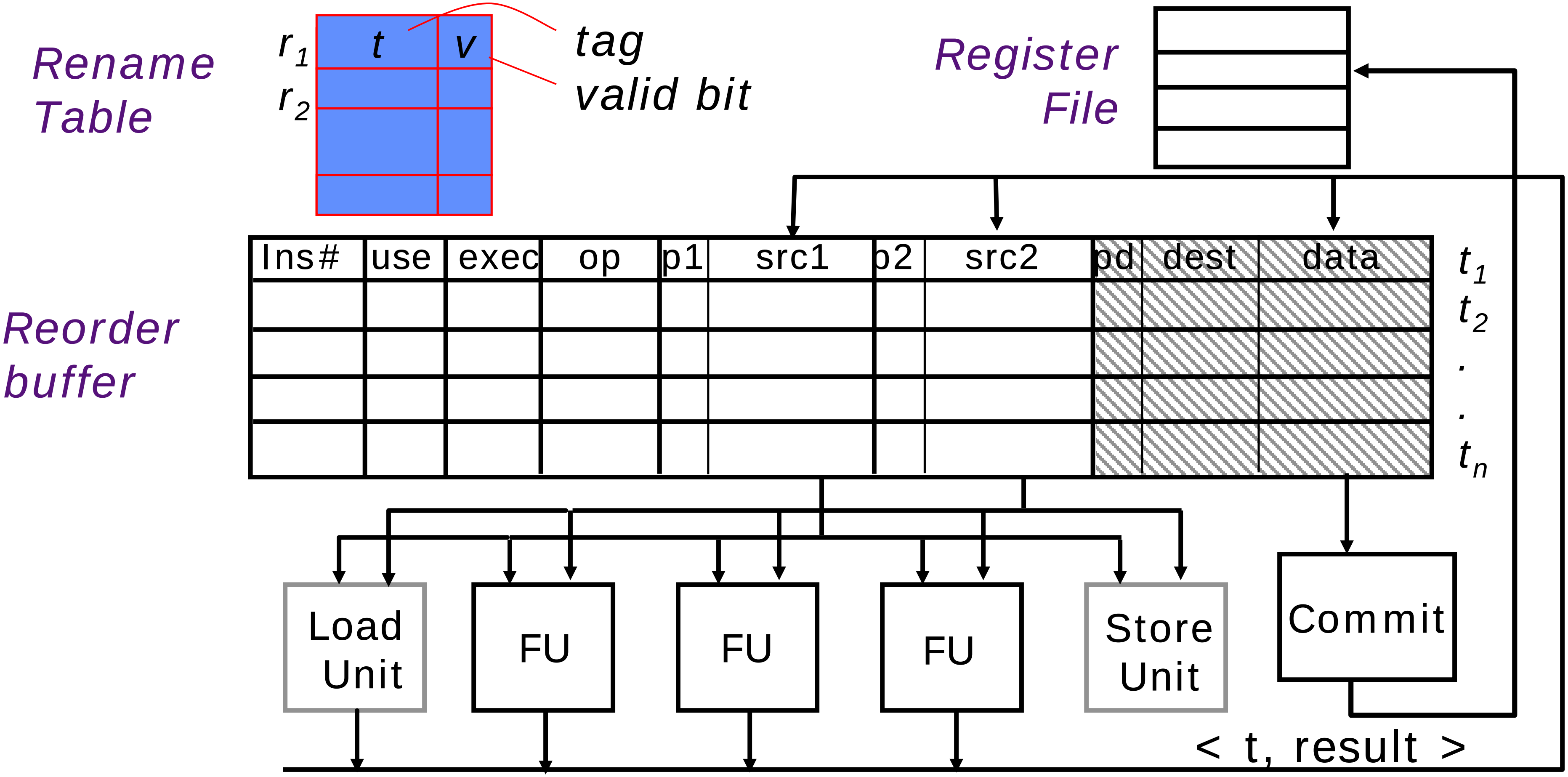
Reorder buffer

Register file does not contain renaming tags any more.

How does the decode stage find the tag of a source register?

Search the “dest” field in the reorder buffer

Renaming Table



Rename Table

r_1 r_2

t *v*

tag

valid bit

Register File

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	pd	dest	data

t_1 t_2 . . t_n

Load Unit

FU

FU

FU

Store Unit

Commit

< t, result >

Renaming table is a cache to speed up register name look up.

It needs to be cleared after each exception taken.

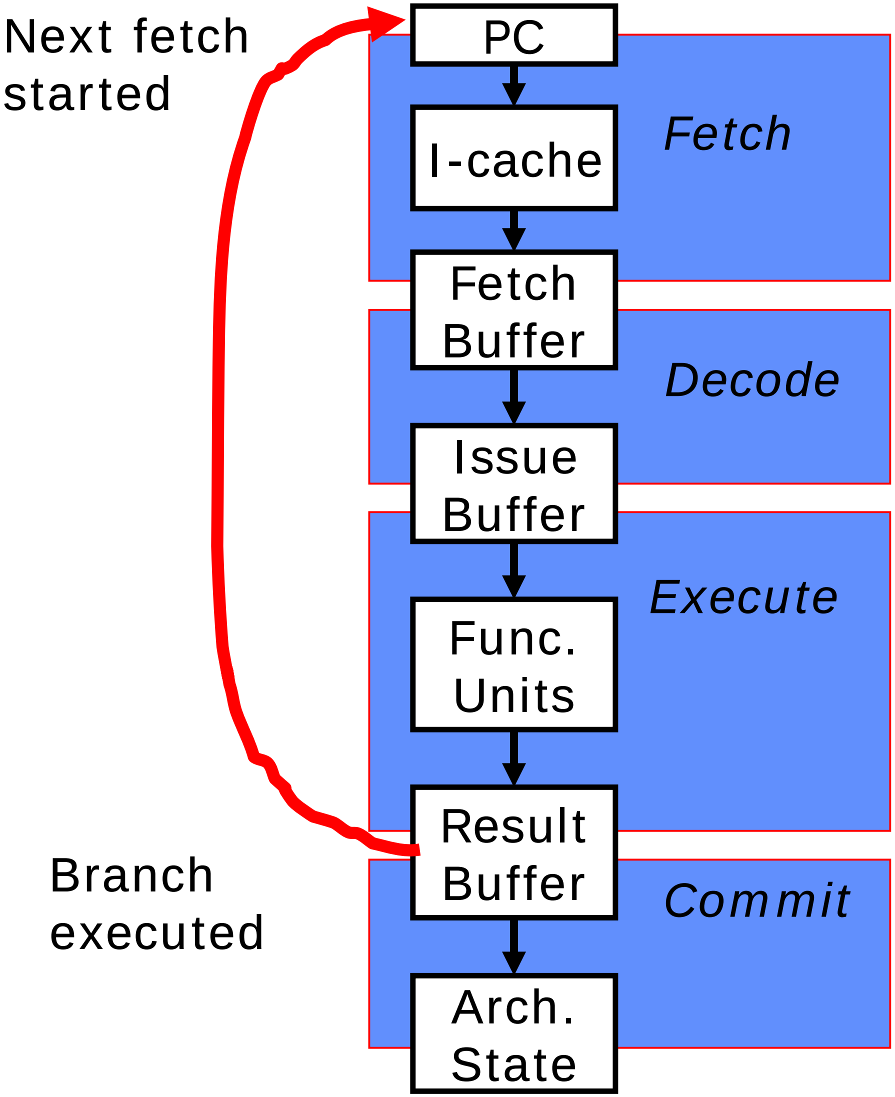
When else are valid bits cleared? *Control transfers*

Control Flow Penalty

Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !

How much work is lost if pipeline doesn't follow correct instruction flow?

~ Loop length x pipeline width



Mispredict Recovery

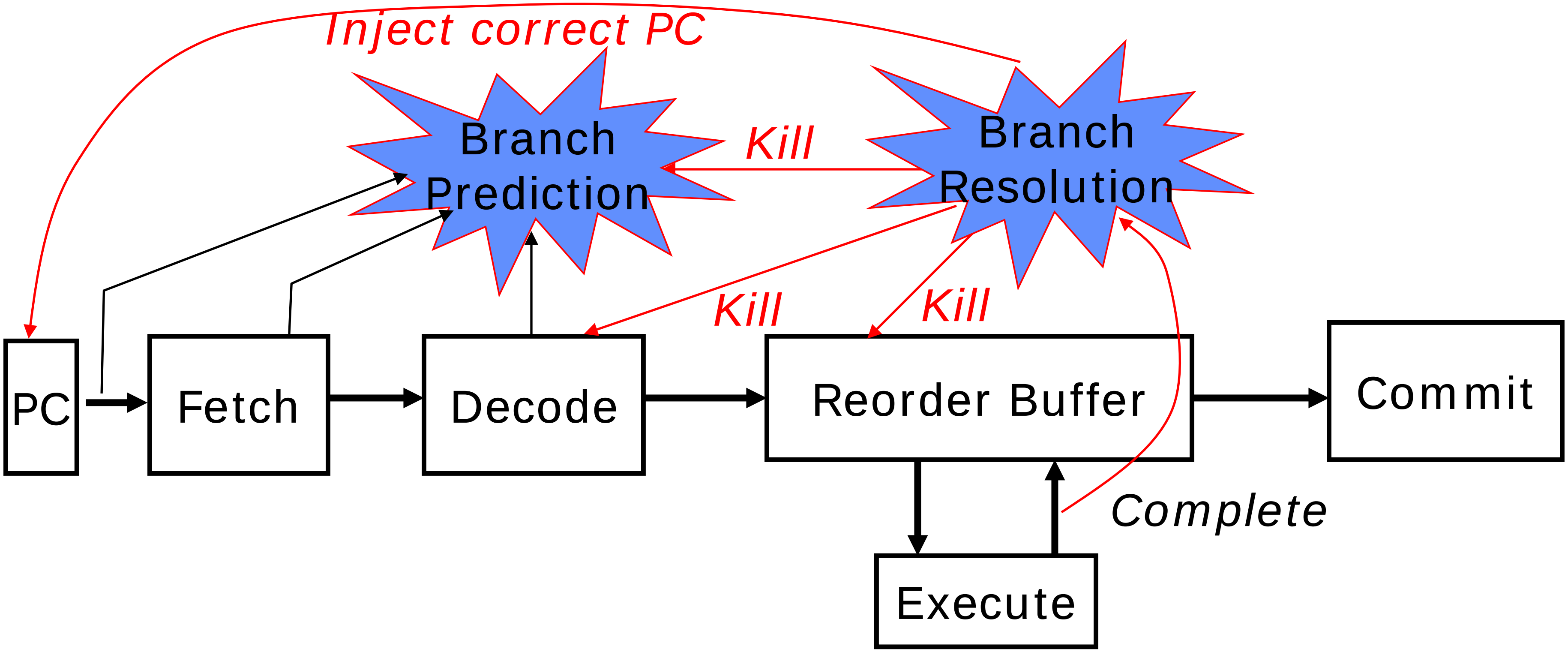
In-order execution machines:

- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution?

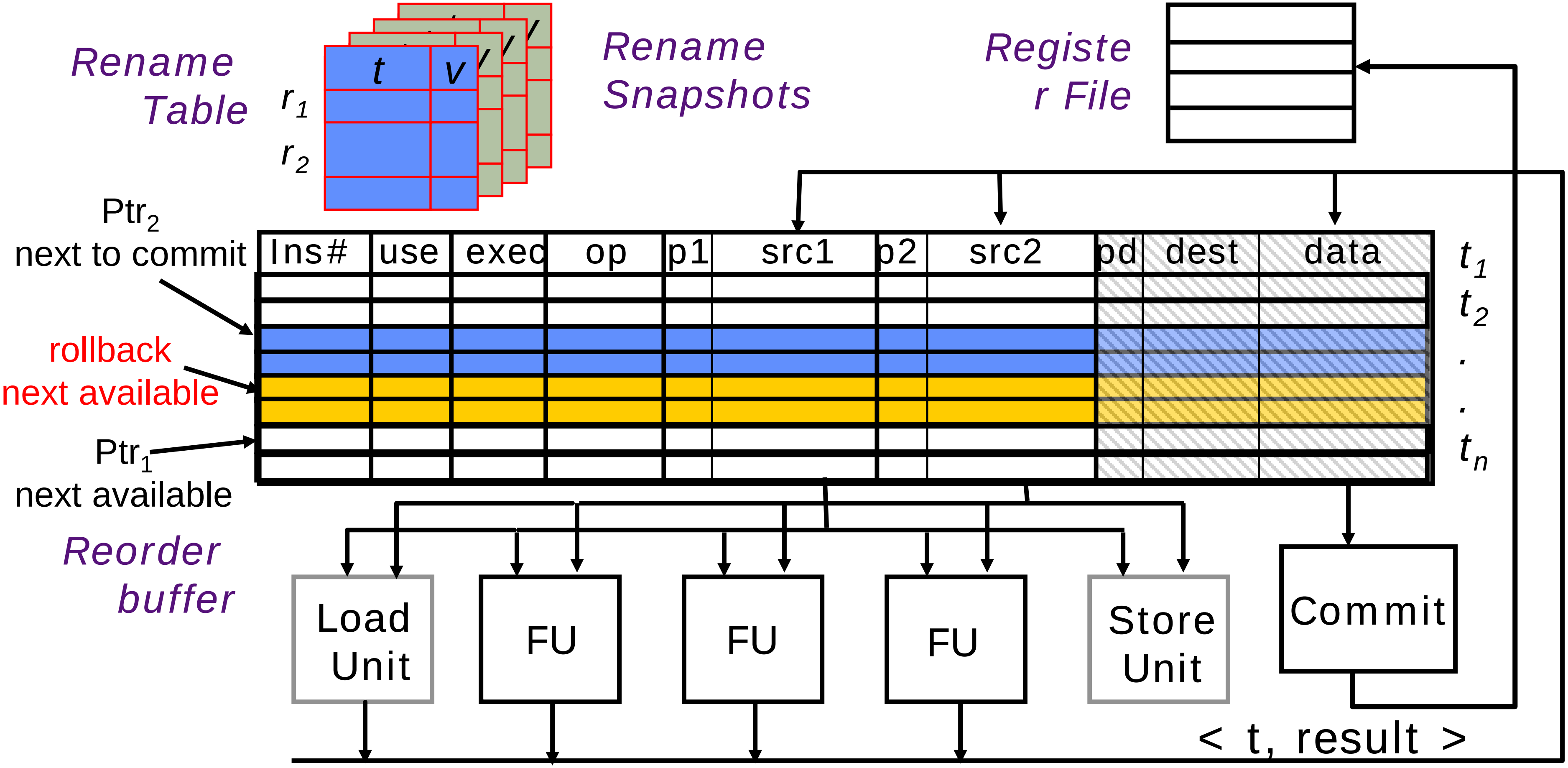
- Multiple instructions following branch in program order can complete before branch resolves

Branch Misprediction in Pipeline



- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch

Recovering ROB/Renaming Table



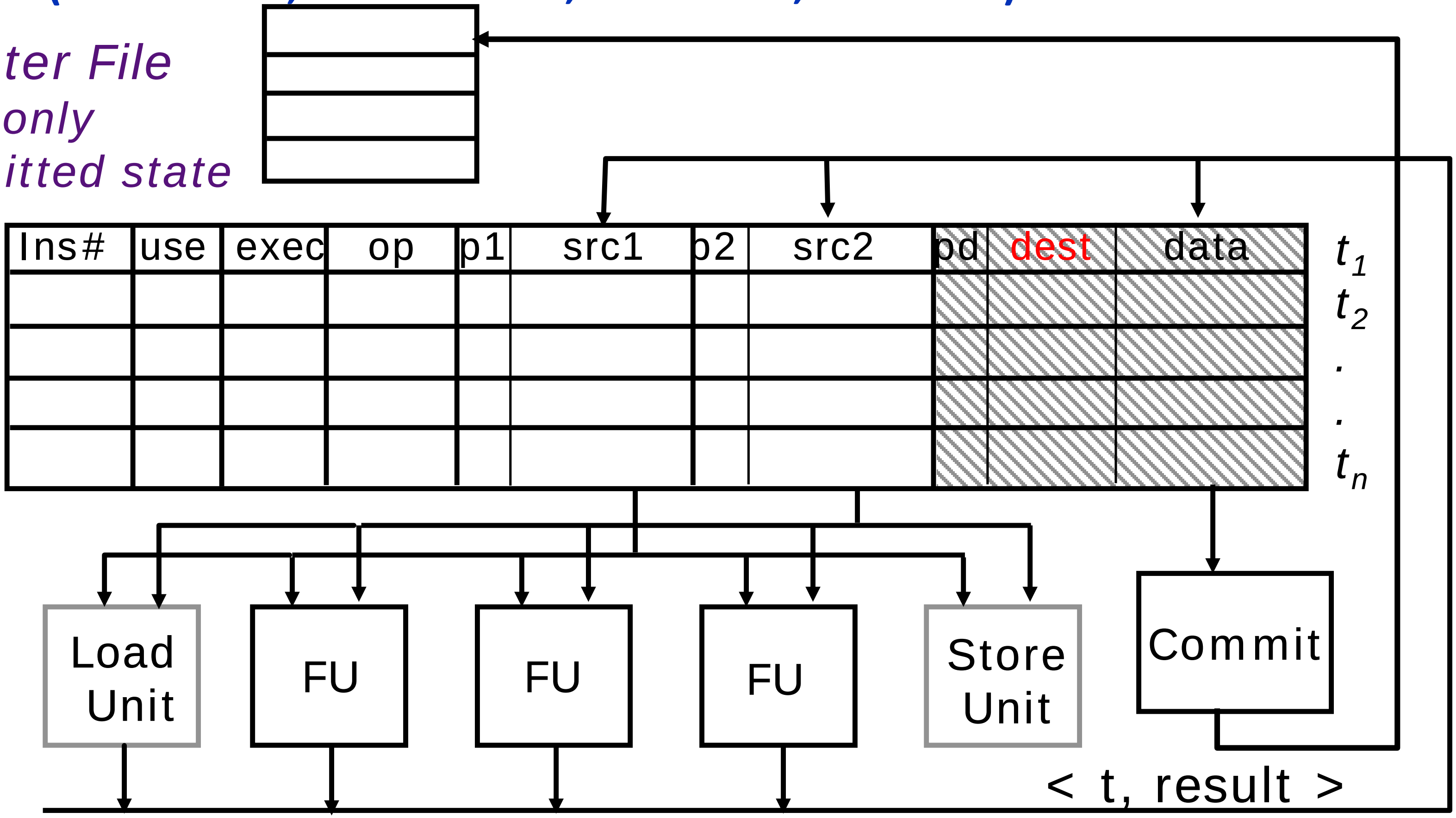
Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted

“Data-in-ROB” Design

(HP PA8000, Pentium Pro, Core2Duo, Nehalem)

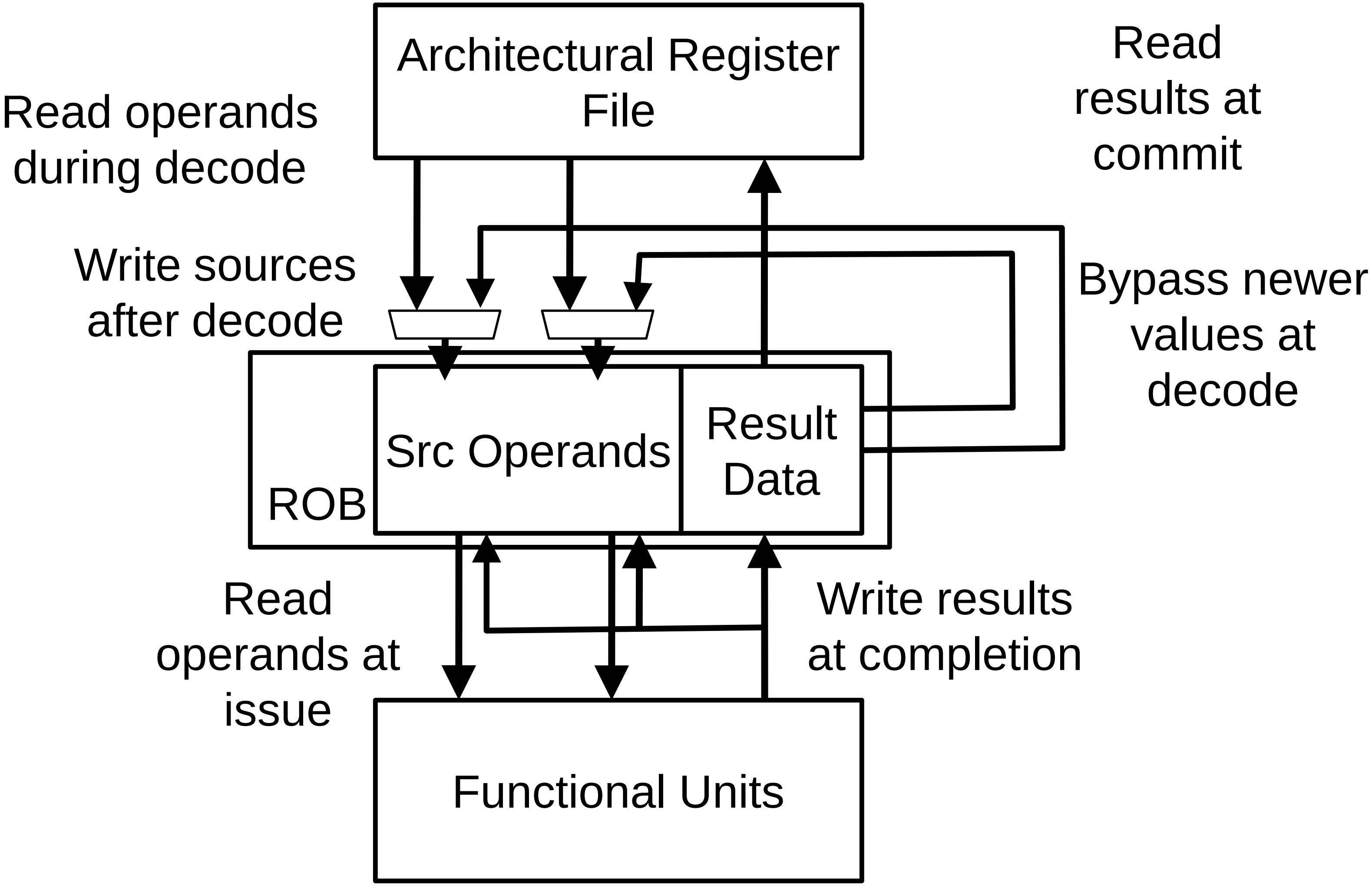
Register File holds only committed state

Reorder buffer



- On dispatch into ROB, ready sources can be in regfile or in ROB dest (copied into src1/src2 if ready before dispatch)
- On completion, write to dest field and broadcast to src fields.
- On issue, read from ROB src fields

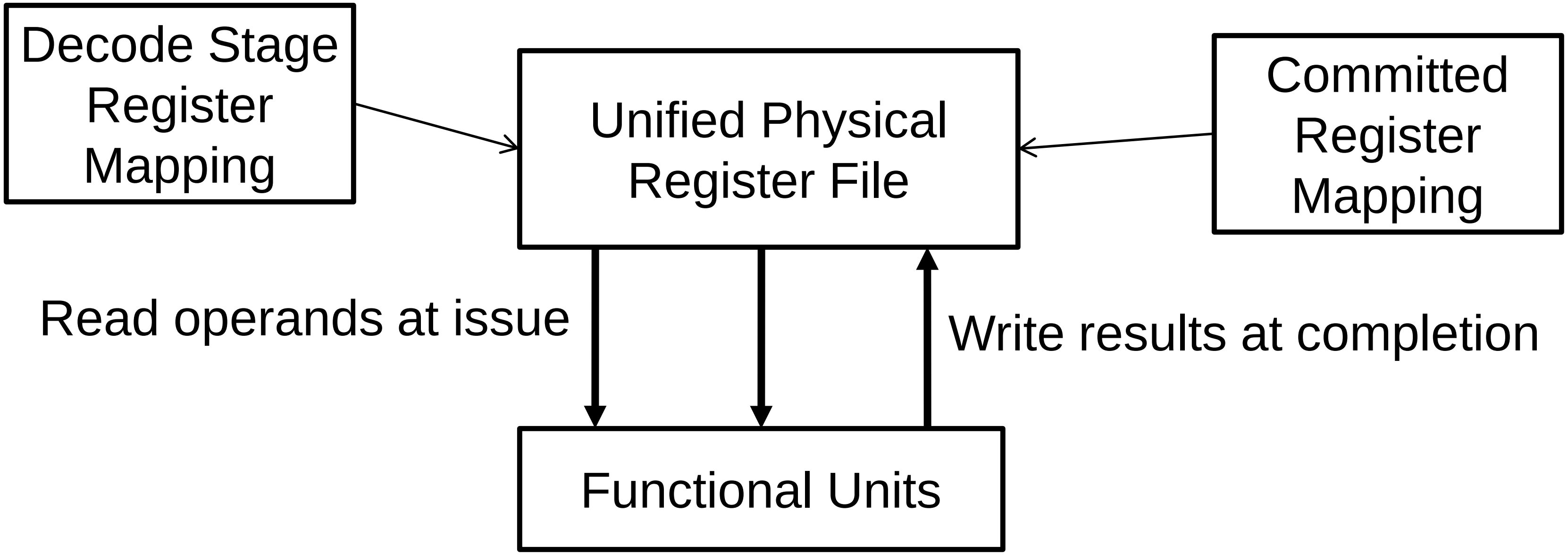
Data Movement in Data-in-ROB Design


Architectural Register File
Read operands during decode
Read results at commit
Write sources after decode
Bypass newer values at decode
Src Operands
Result Data
ROB
Read operands at issue
Write results at completion
Functional Units

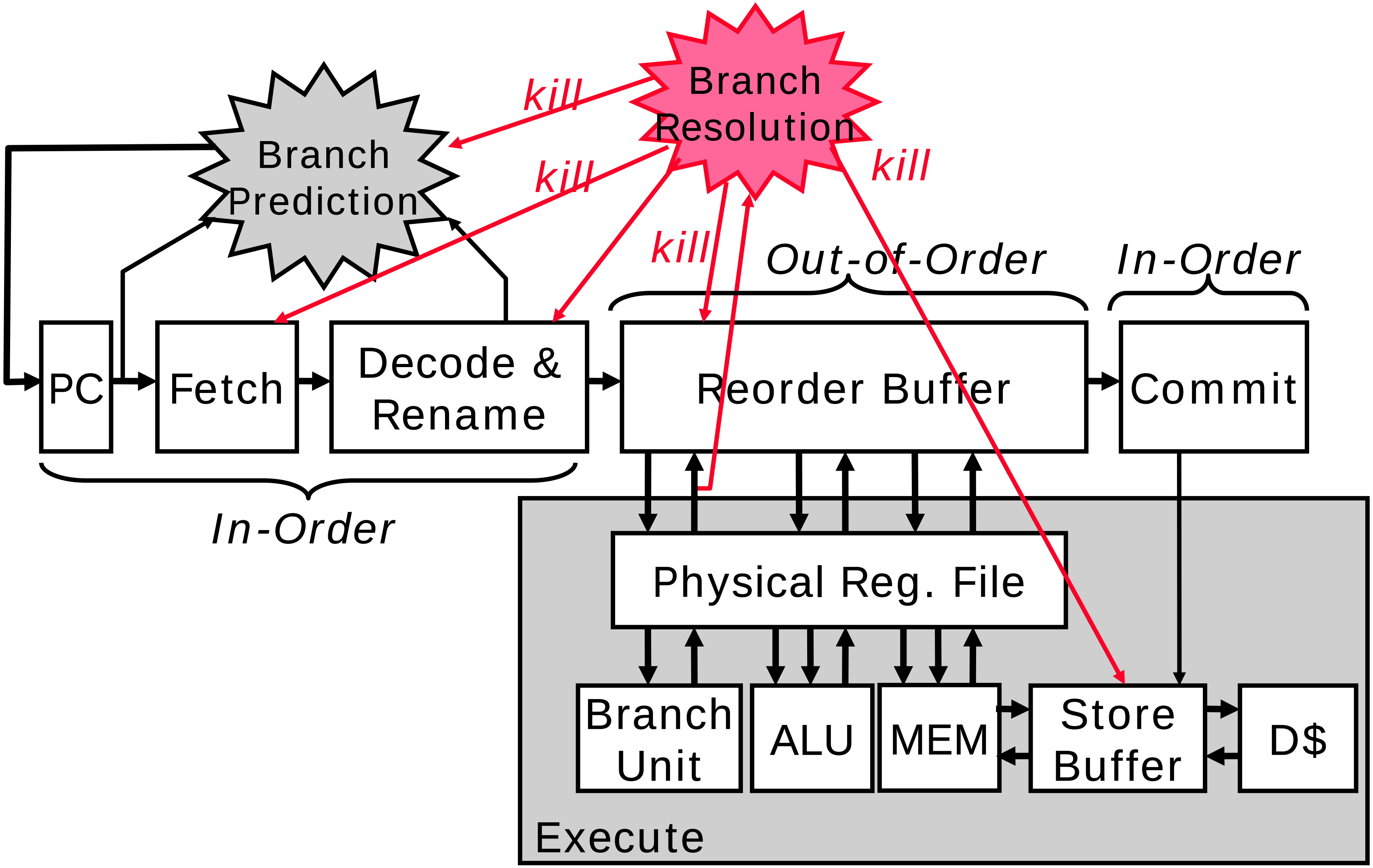
Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy Bridge)

- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical register, no data movement


Decode Stage Register Mapping
Unified Physical Register File
Committed Register Mapping
Read operands at issue
Write results at completion
Functional Units

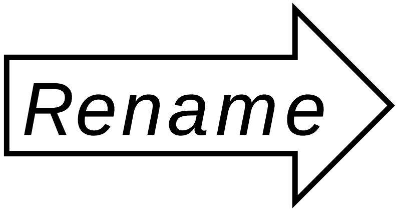
Pipeline Design with Physical Regfile



Lifetime of Physical Registers

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*

```
ld x1, (x3)
addi x3, x1, #4
sub x6, x7, x9
add x3, x3, x6
ld x6, (x1)
add x6, x6, x3
sd x6, (x1)
ld x6, (x11)
```



Rename

```
ld P1, (Px)
addi P2, P1, #4
sub P3, Py, Pz
add P4, P2, P3
ld P5, (P1)
add P6, P5, P4
sd P6, (P1)
ld P7, (Pw)
```

When can we reuse a physical register?

When next write of same architectural register commits

Physical Register Management

Rename Table

x0	
x1	P8
x2	
x3	P7
x4	
x5	
x6	P5
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<x1>	p
Pn		

Free List

P0
P1
P3
P2
P4

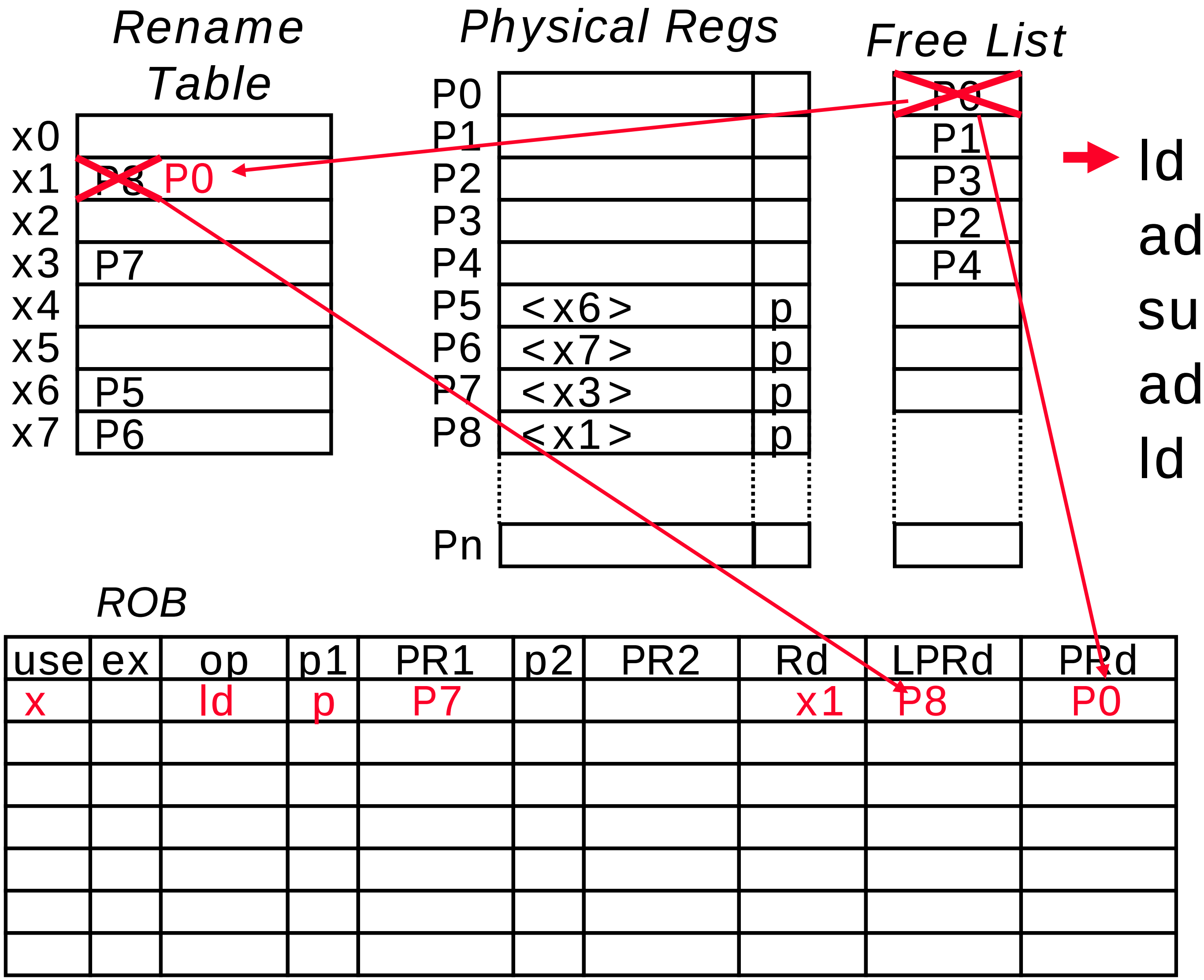
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

(LPRd requires third read port on Rename Table for each instruction)

Physical Register Management



Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	P7
x4	
x5	
x6	P5
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<x1>	p
Pn		

Free List

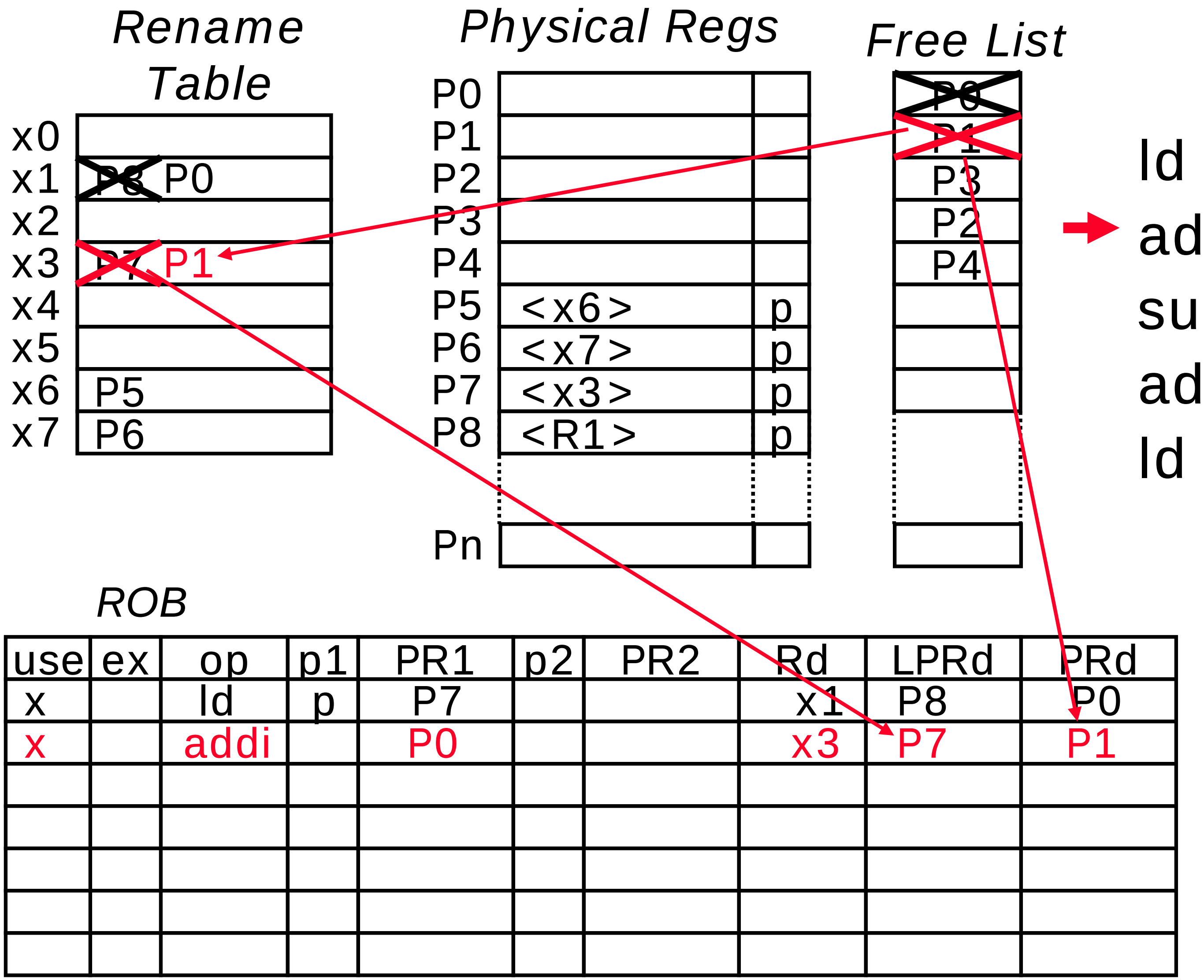
~~P0~~
P1
P3
P2
P4

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0

```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

Physical Register Management



Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ P1
x4	
x5	
x6	P5
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<R1>	p
Pn		

Free List

~~P0~~
~~P1~~
P3
P2
P4

ld x1, 0(x3)

→ addi x3, x1, #4

sub x6, x7, x6

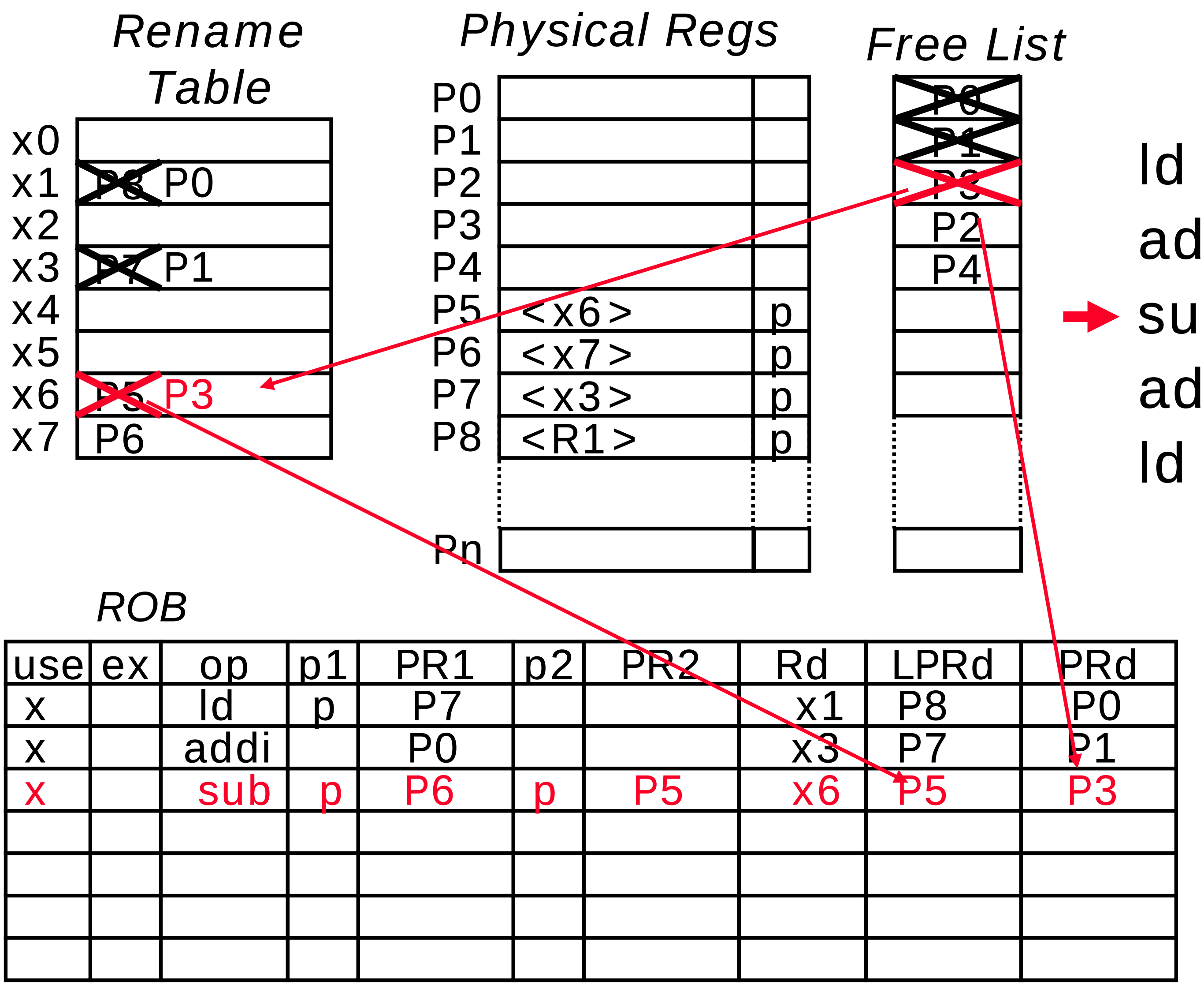
add x3, x3, x6

ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1

Physical Register Management



Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ P1
x4	
x5	
x6	~~P5~~ P3
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<R1>	p
Pn		

Free List

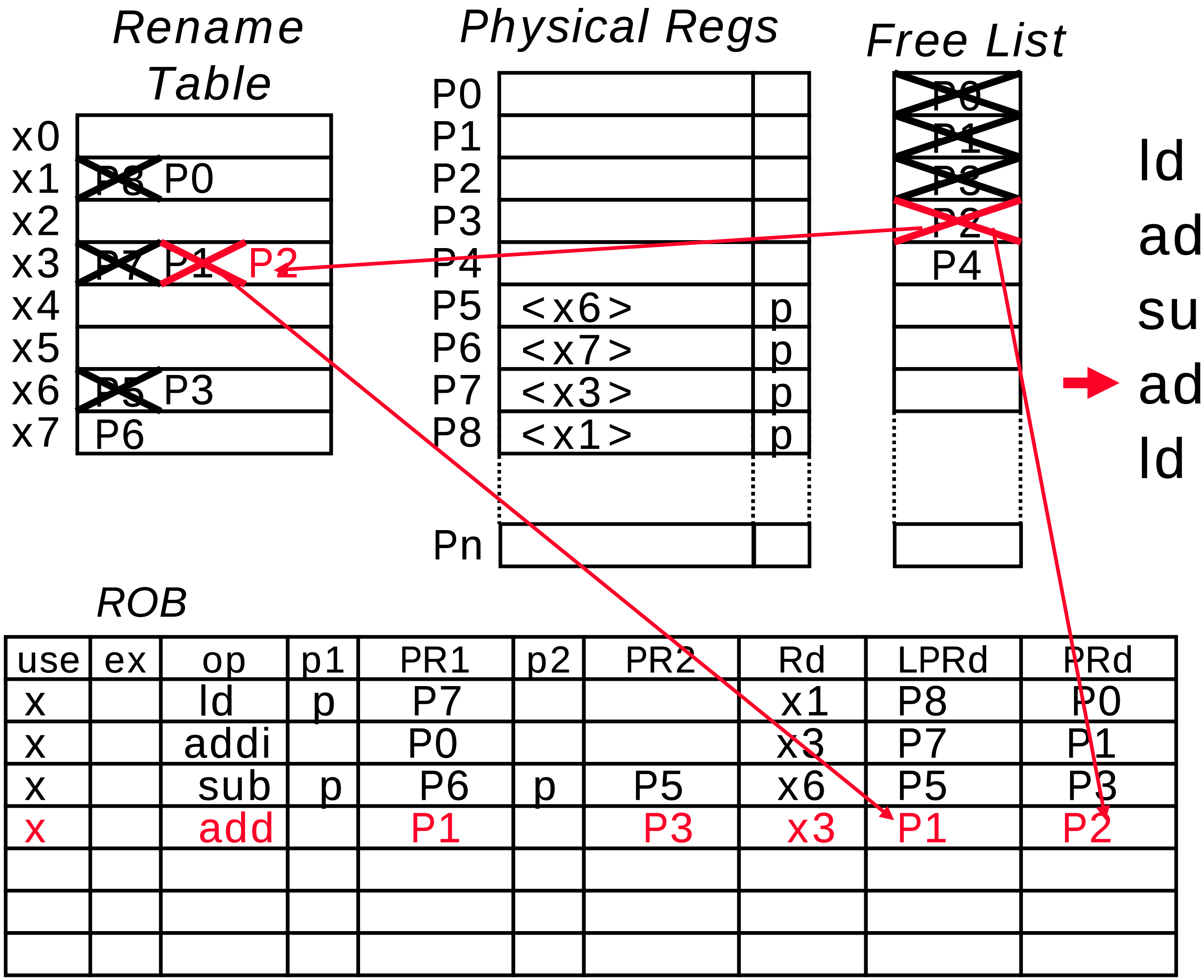
~~P0~~
~~P1~~
~~P3~~
P2
P4

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3

ld x1, 0(x3)
addi x3, x1, #4
→ sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

Physical Register Management



Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ ~~P1~~ P2
x4	
x5	
x6	~~P5~~ P3
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<x1>	p
Pn		

Free List

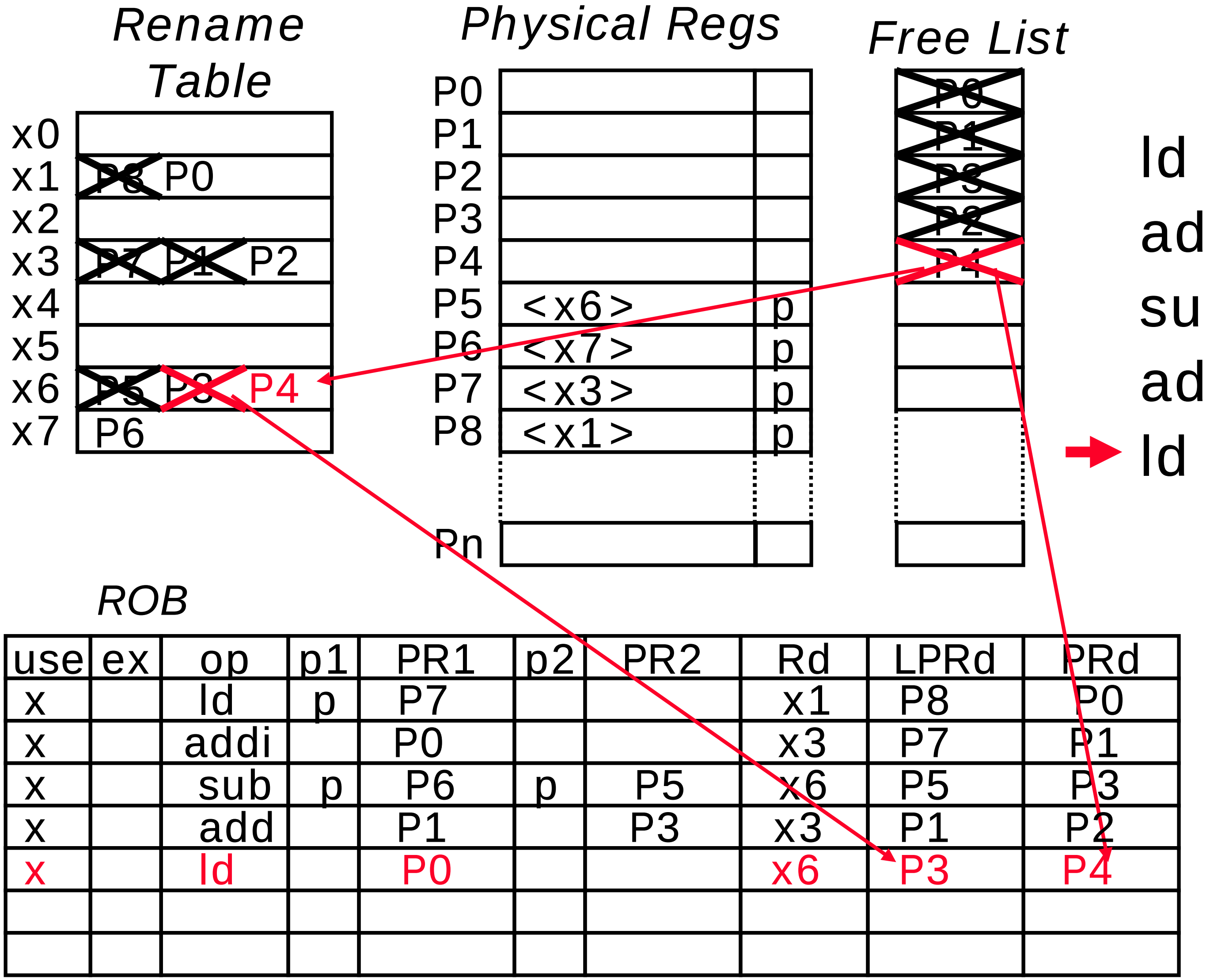
~~P0~~
~~P1~~
~~P3~~
~~P2~~
P4

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2

ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
➡ add x3, x3, x6
ld x6, 0(x1)

Physical Register Management



Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ ~~P1~~ P2
x4	
x5	
x6	~~P5~~ ~~P3~~ P4
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<x1>	p
Pn		

Free List

~~P0~~
~~P1~~
~~P3~~
~~P2~~
~~P4~~

ld x1, 0(x3)

addi x3, x1, #4

sub x6, x7, x6

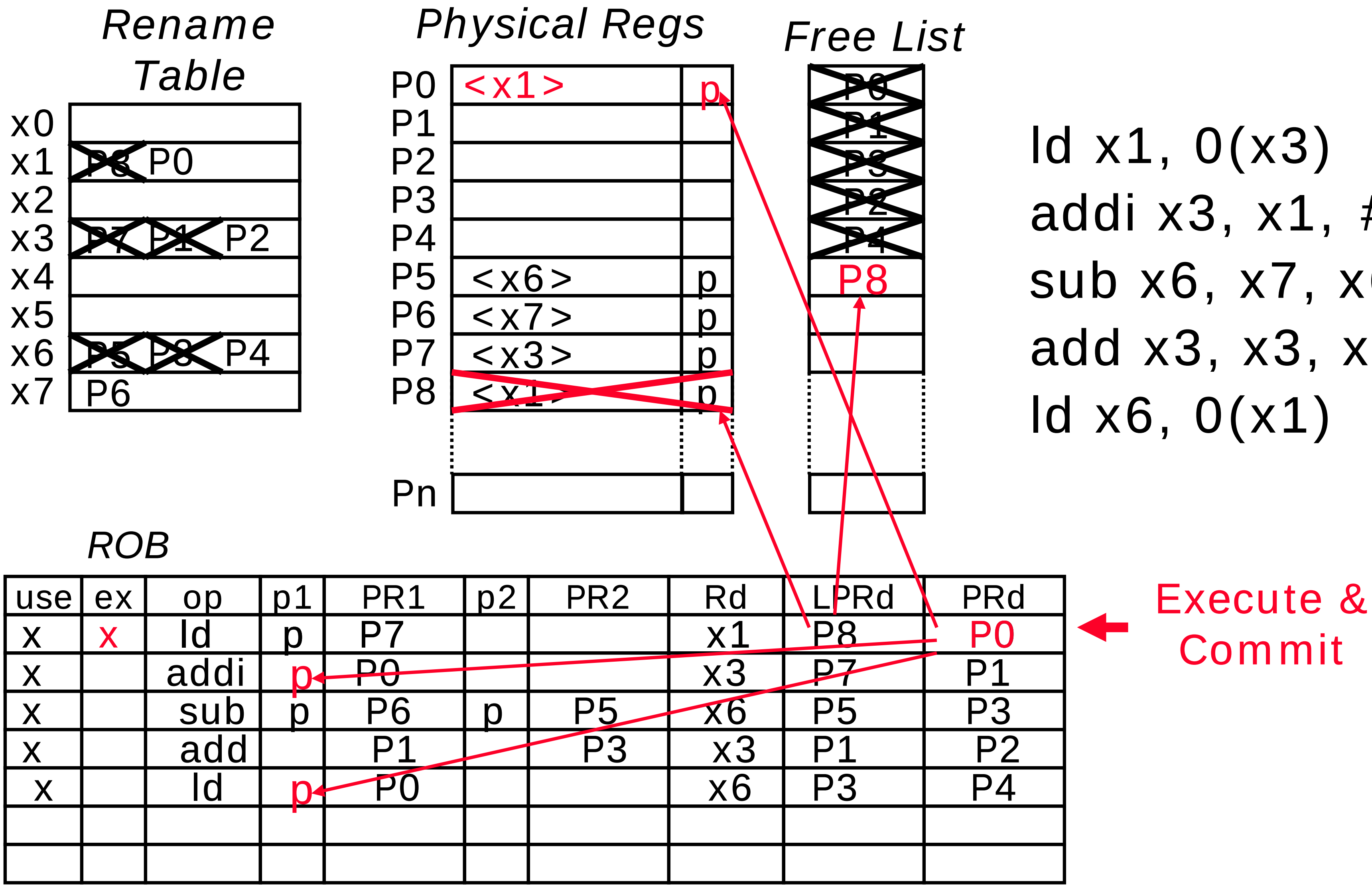
add x3, x3, x6

→ ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld		P0			x6	P3	P4

Physical Register Management



Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ ~~P1~~ P2
x4	
x5	
x6	~~P5~~ ~~P3~~ P4
x7	P6

Physical Regs

P0	<x1>	p
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	~~<x1>~~	~~p~~
Pn		

Free List

~~P0~~
~~P1~~
~~P3~~
~~P2~~
~~P4~~
P8

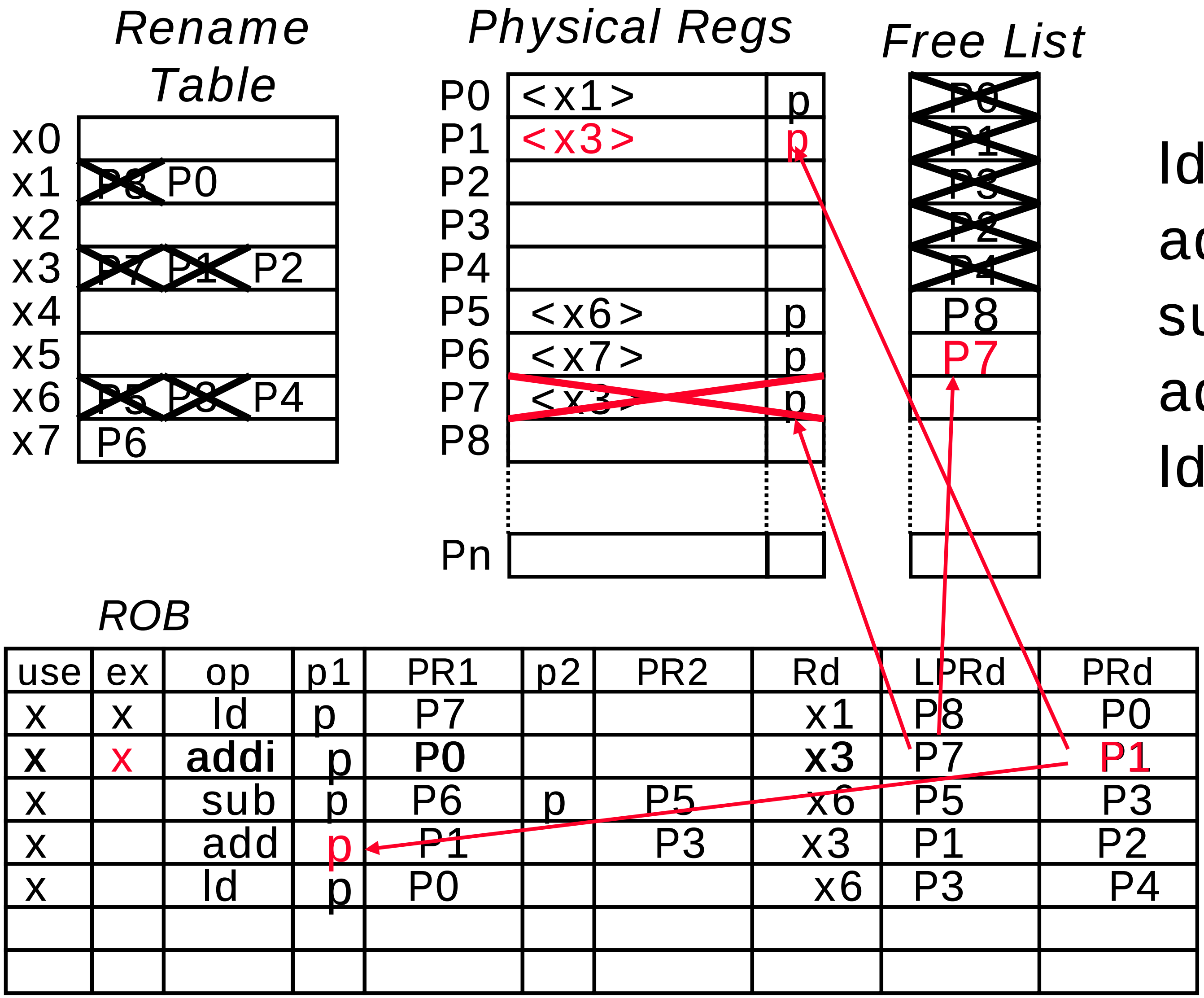
ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x		addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

Execute & Commit

ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

Physical Register Management



Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ ~~P1~~ P2
x4	
x5	
x6	~~P5~~ ~~P3~~ P4
x7	P6

Physical Regs

P0	<x1>	p
P1	<x3>	p
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	~~<x3>~~	~~p~~
P8		
Pn		

Free List

~~P0~~
~~P1~~
~~P3~~
~~P2~~
~~P4~~
P8
P7

ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x	x	**addi**	p	**P0**			**x3**	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add	p	P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

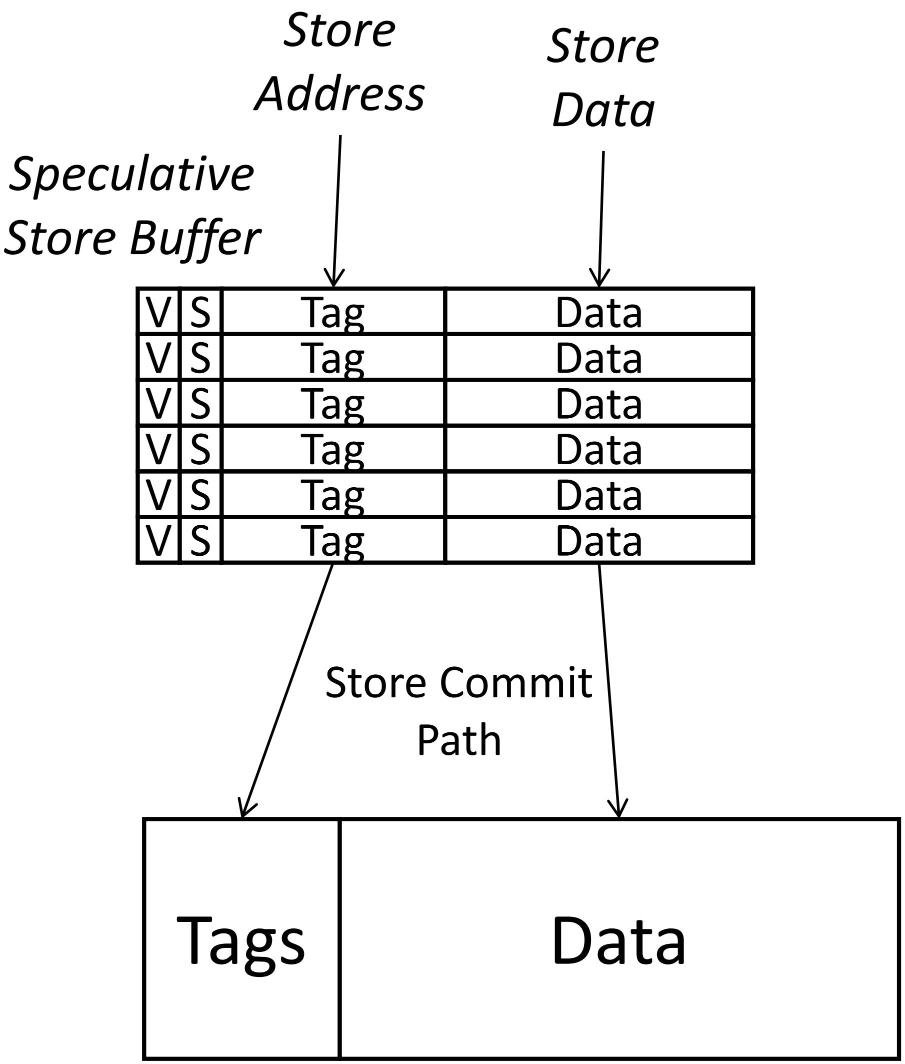
Execute & Commit

Speculative Loads / Stores

Just like register updates, stores should not modify the memory until after the instruction is committed

- A speculative store buffer is a structure introduced to hold speculative store data.

Speculative Store Buffer



Store Address

Store Data

Speculative Store Buffer

V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data

Store Commit Path

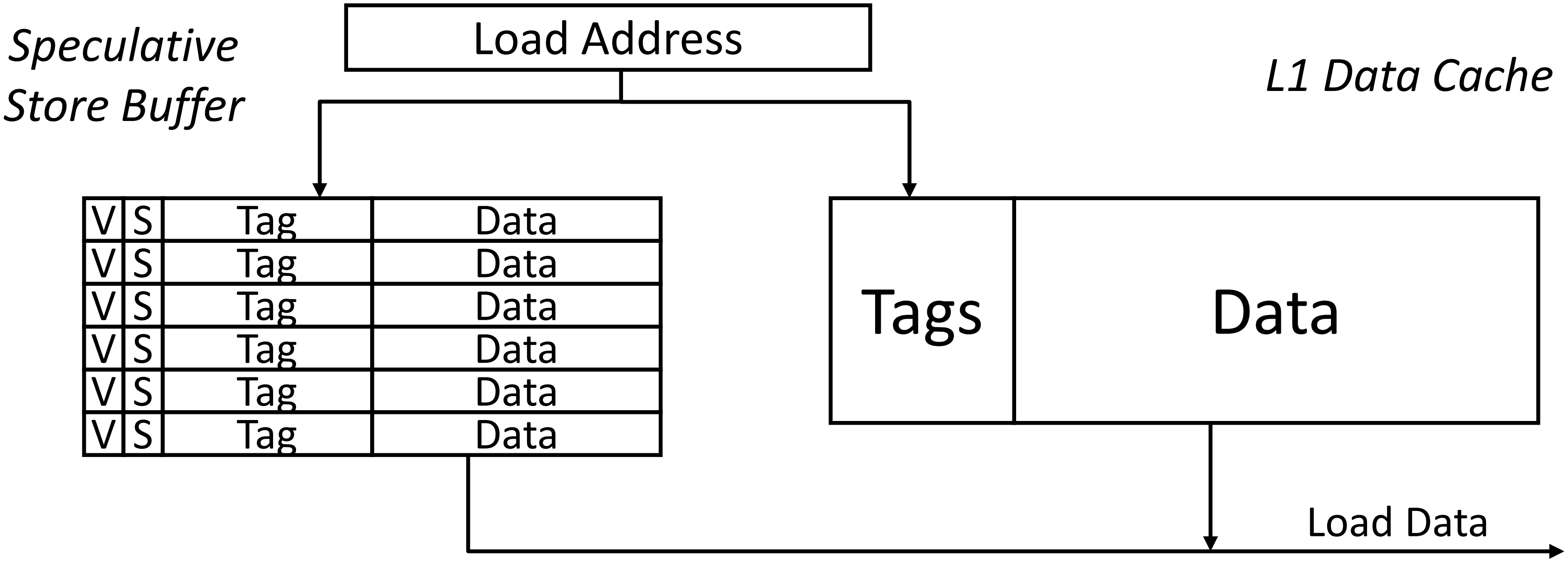
Tags	Data

L1 Data Cache

Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.

- During decode, store buffer slot allocated in program order
- Stores split into “store address” and “store data” micro-operations
- “Store address” execute writes tag
- “Store data” execute writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Load bypass from speculative store buffer



- If data in both store buffer and cache, which should we use?

  Speculative store buffer

- If same address in store buffer twice, which should we use?

  Youngest store older than load

Memory Dependencies

```
sd x1, (x2)
ld x3, (x4)
```

When can we execute the load?

In-Order Memory Queue

- Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

- Need a structure to handle memory ordering…

Conservative O-o-O Load Execution

```
sd x1, (x2)
ld x3, (x4)
```

- Can execute load before store, if addresses known and x4 != x2
- Each load address compared with addresses of all previous uncommitted stores
  - *can use partial conservative check i.e., bottom 12 bits of address, to save hardware*
- Don’t execute load if any previous store address not known

(MIPS R10K, 16-entry address queue)

Address Speculation

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that x4 != x2

- Execute load before store address known

- Need to hold all completed but uncommitted load/store addresses in program order

- If subsequently find x4==x2, squash load and *all* following instructions

=> Large penalty for inaccurate address speculation

Memory Dependence Prediction

(Alpha 21264)

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that x4 != x2 and execute load before store
- If later find x4==x2, squash load and all following instructions, but mark load instruction as *store-wait*
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear *store-wait* bits

Recap: Dynamic Scheduling

- Three big ideas: register renaming, data-driven detection of RAW resolution, speculative execution with in-order commit .
- Very complex, but enables many things: out-of-order execution, multiple issue, loop unrolling, etc.
- Has saved architectures that have a small number of registers: IBM 360 floating-point ISA, Intel x86 ISA.

Chapter 5 (2)

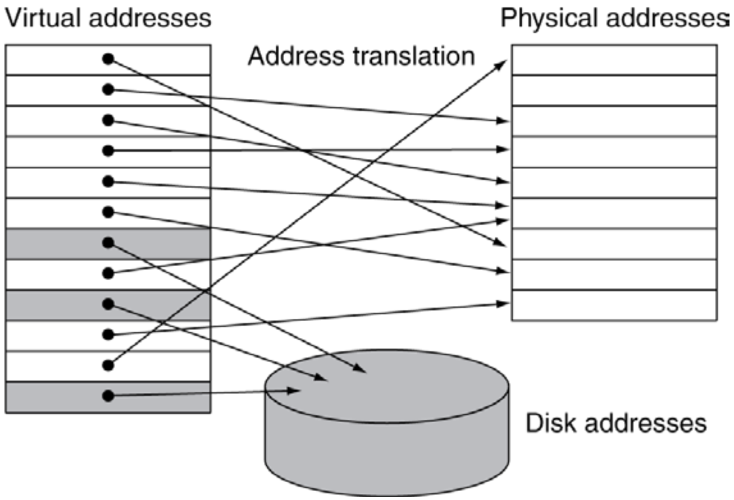
Virtual Memory & Advanced Memory Hierarchy

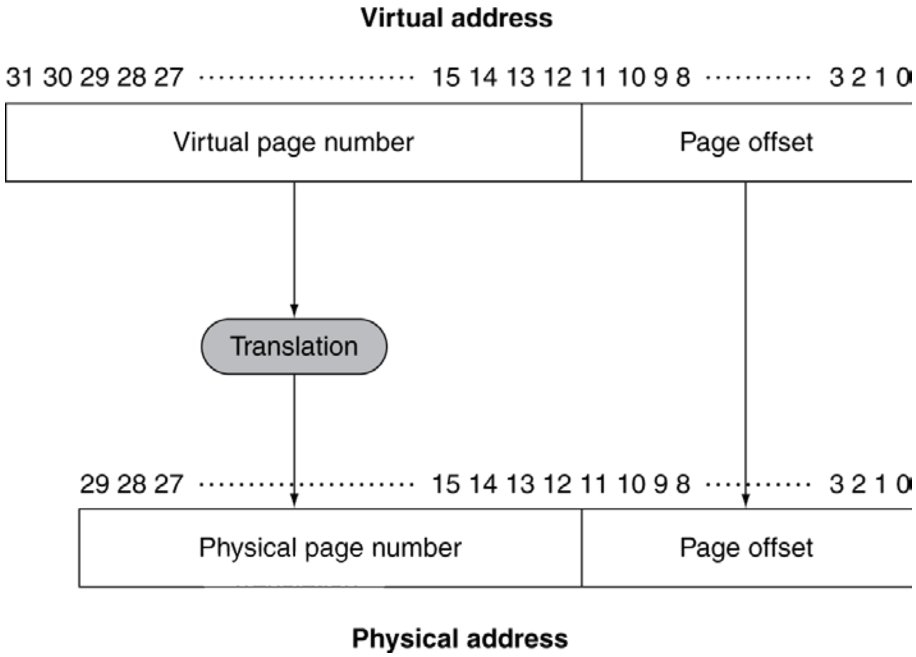
Virtual Memory

- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM “block” is called a page
  - VM translation “miss” is called a page fault

Address Translation

- Fixed-size pages (e.g., 4K)





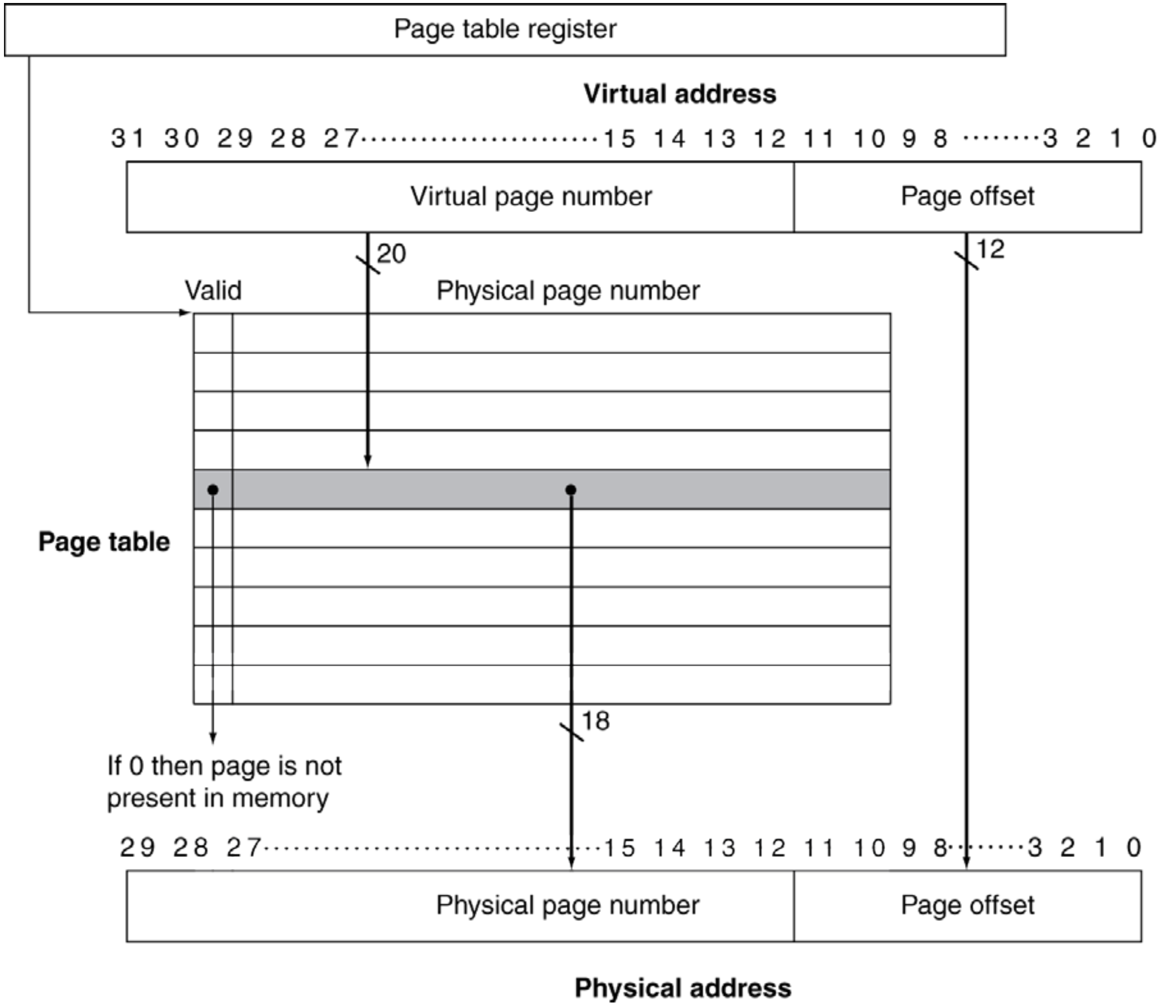
Page Fault Penalty

- On page fault, the page must be fetched from disk
  - Takes millions of clock cycles
  - Handled by OS code
- Try to minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms

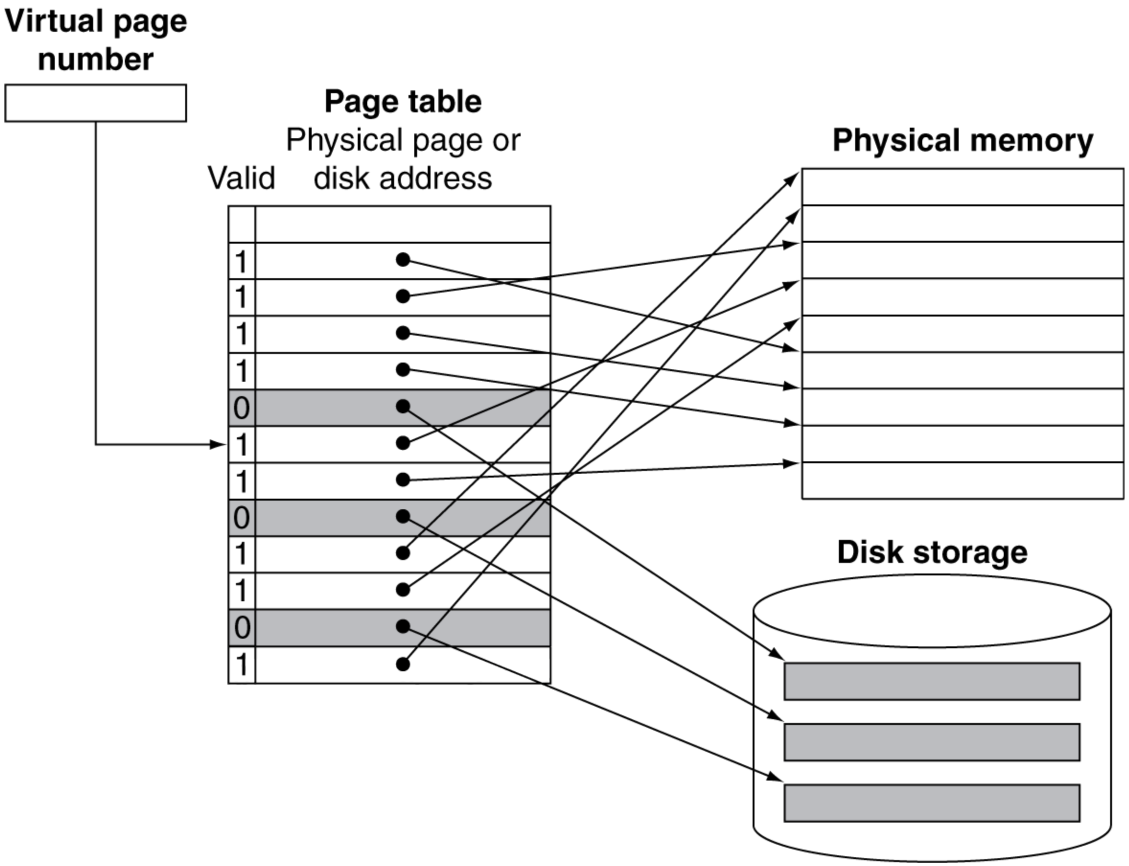
Page Tables

- Stores placement information
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory
- If page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- If page is not present
  - PTE can refer to location in swap space on disk

Translation Using a Page Table



Mapping Pages to Storage



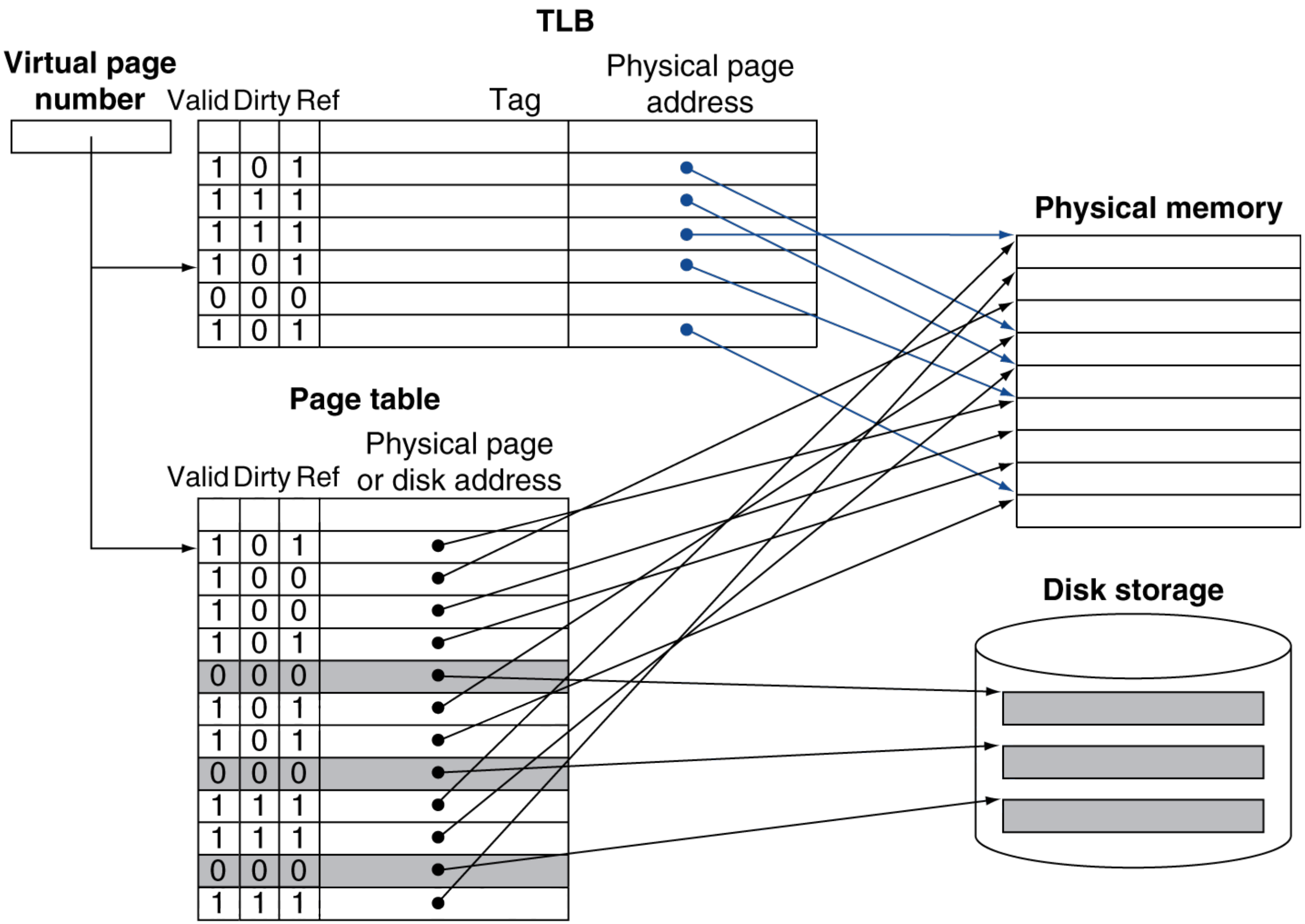
Replacement and Writes

- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
- Disk writes take millions of cycles
  - Block at once, not individual locations
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

Fast Translation Using a TLB

- Address translation would appear to require extra memory references
  - One to access the PTE
  - Then the actual memory access
- But access to page tables has good locality
  - So use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software

Fast Translation Using a TLB



TLB

Valid	Dirty	Ref	Tag	Physical page address
1	0	1		●
1	1	1		●
1	1	1		●
1	0	1		●
0	0	0		
1	0	1		●

Page table

Valid	Dirty	Ref	Physical page or disk address
1	0	1	●
1	0	0	●
1	0	0	●
1	0	1	●
0	0	0	●
1	0	1	●
1	0	1	●
0	0	0	●
1	1	1	●
1	1	1	●
0	0	0	●
1	1	1	●

TLB Misses

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

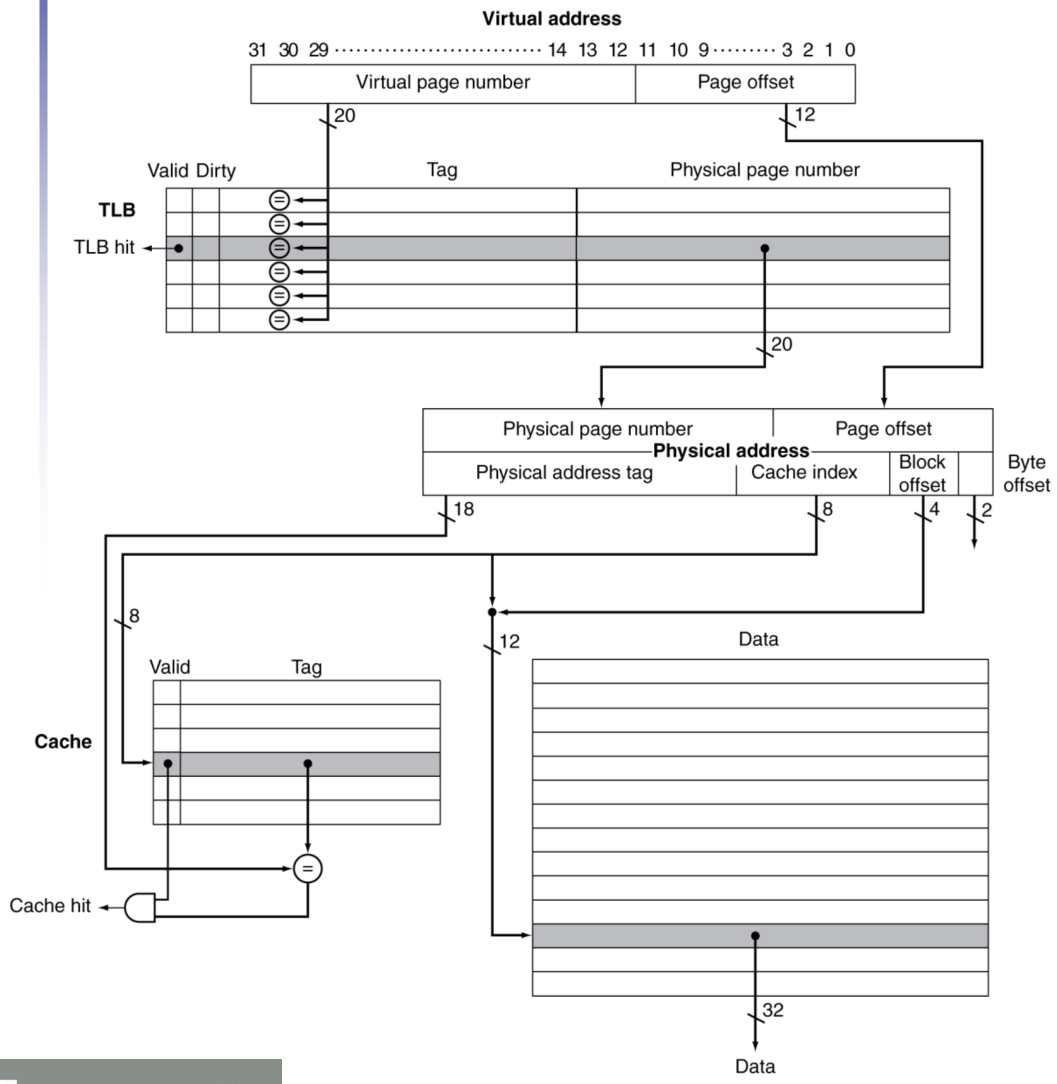
TLB Miss Handler

- TLB miss indicates
  - Page present, but PTE not in TLB
  - Page not preset
- Must recognize TLB miss before destination register overwritten
  - Raise exception
- Handler copies PTE from memory to TLB
  - Then restarts instruction
  - If page not present, page fault will occur

Page Fault Handler

- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
  - Restart from faulting instruction

TLB and Cache Interaction

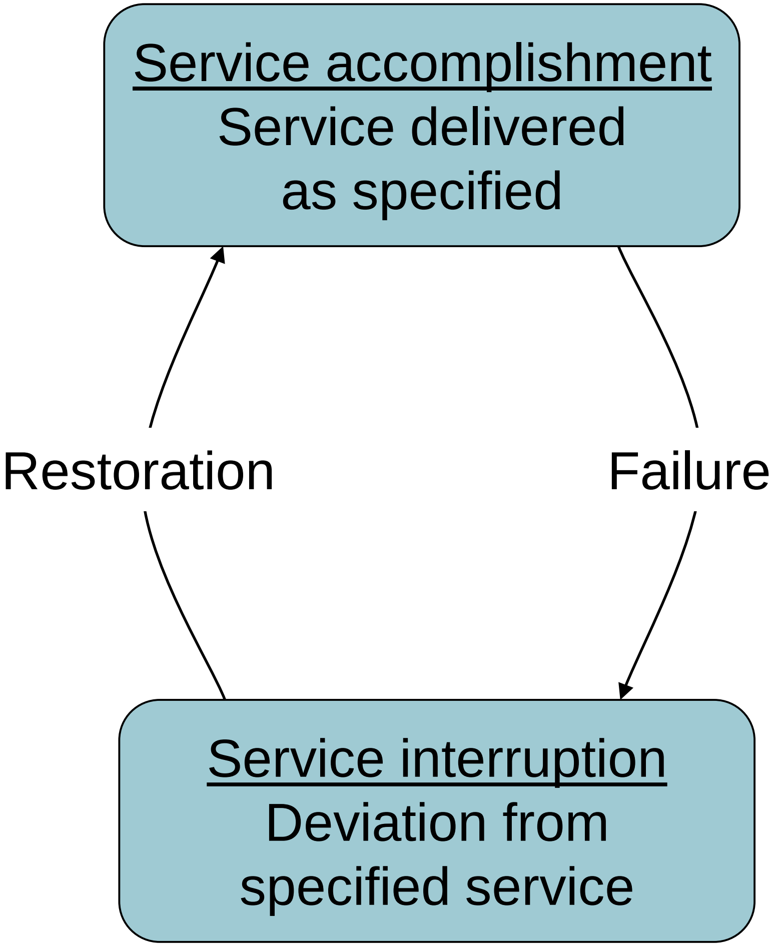


- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

Memory Protection

- Different tasks can share parts of their virtual address spaces
  - But need to protect against errant access
  - Requires OS assistance
- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS)

Dependability



- Fault: failure of a component
  - May or may not lead to system failure

Dependability Measures

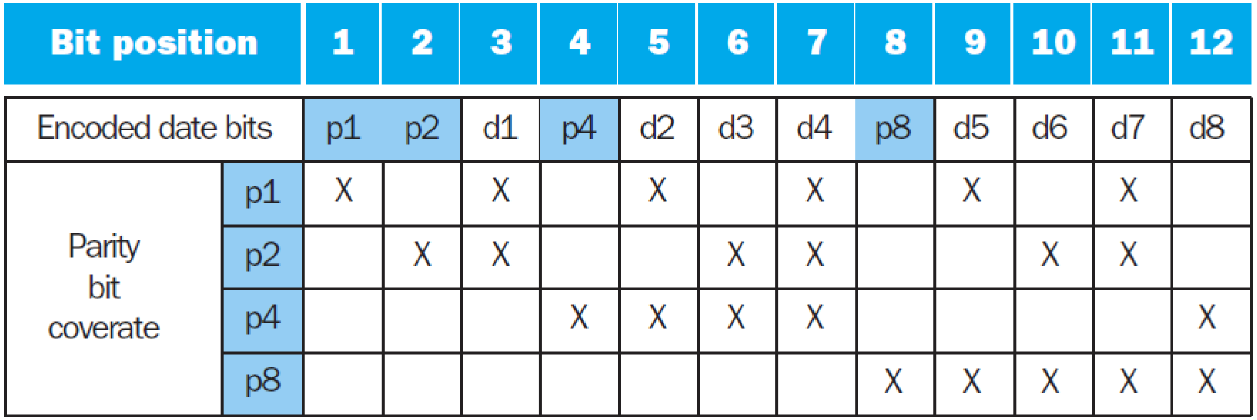
- Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

The Hamming SEC Code

- Hamming distance
  - Number of bits that are different between two bit patterns
- Minimum distance = 2 provides single bit error detection
  - E.g. parity code
- Minimum distance = 3 provides single error correction, 2 bit error detection

Encoding SEC

- To calculate Hamming code:
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - Each parity bit checks certain data bits:



Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverate	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

Decoding SEC

- Value of parity bits indicates which bits are in error
  - Use numbering from encoding procedure
  - E.g.
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 indicates bit 10 was flipped

SEC/DEC Code

- Add an additional parity bit for the whole word (p_n)
- Make Hamming distance = 4
- Decoding:
  - Let H = SEC parity bits
    - H even, p_n even, no error
    - H odd, p_n odd, correctable single bit error
    - H even, p_n odd, error in p_n bit
    - H odd, p_n even, double error occurred
- Note: ECC DRAM uses SEC/DEC with 8 bits protecting each 64 bits

Optimization: basic

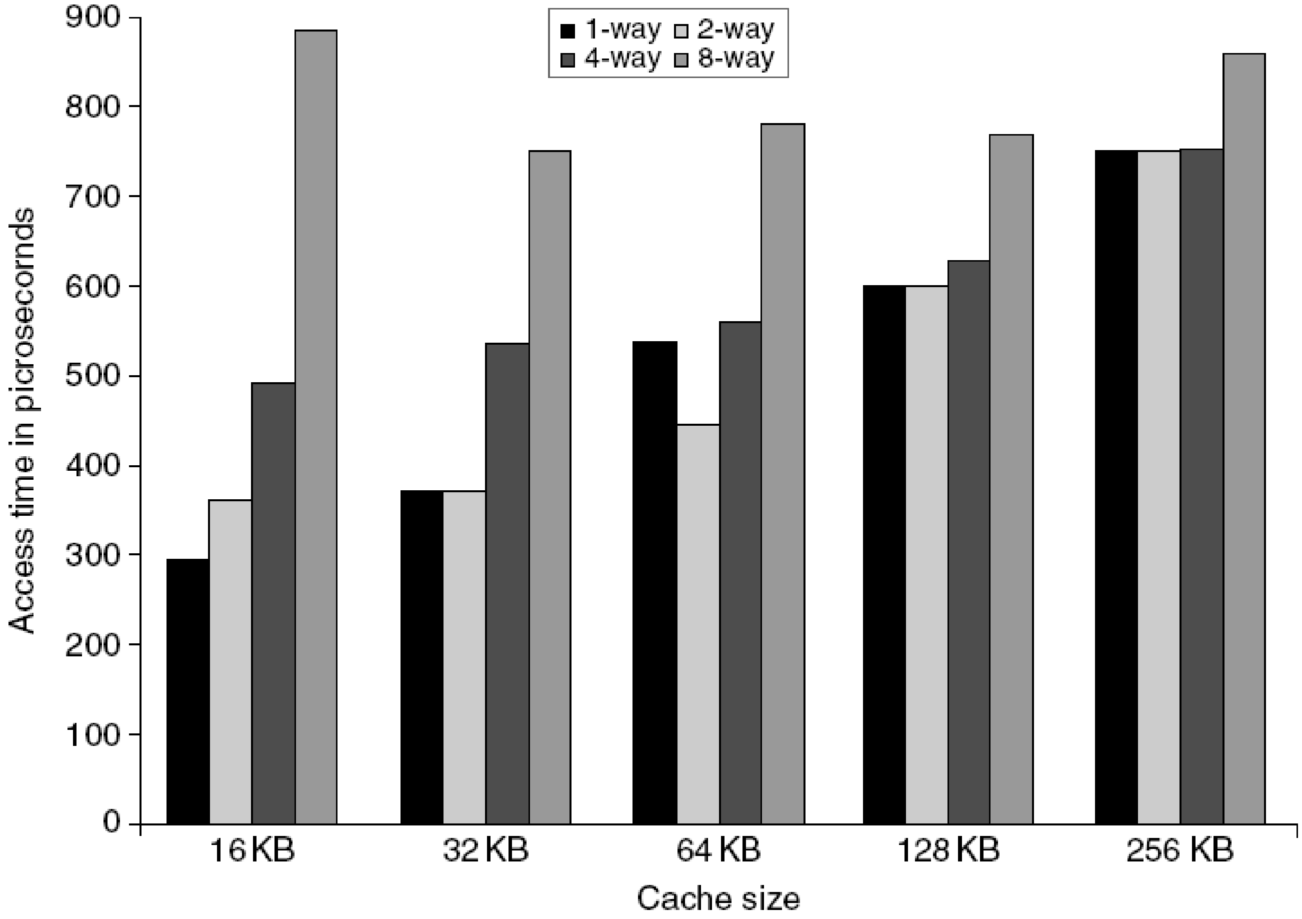
- Six basic cache optimizations:
  - Larger block size
    - Reduces compulsory misses
    - Increases capacity and conflict misses, increases miss penalty
  - Larger total cache capacity to reduce miss rate
    - Increases hit time, increases power consumption
  - Higher associativity
    - Reduces conflict misses
    - Increases hit time, increases power consumption
  - Higher number of cache levels
    - Reduces overall memory access time
  - Giving priority to read misses over writes
    - Reduces miss penalty
  - Avoiding address translation in cache indexing
    - Reduces hit time

Optimization: advanced

- Small and simple first level caches
  - Critical timing path:
    - addressing tag memory, then
    - comparing tags, then
    - selecting correct set
  - Direct-mapped caches can overlap tag compare and transmission of data
  - Lower associativity reduces power because fewer cache lines are accessed

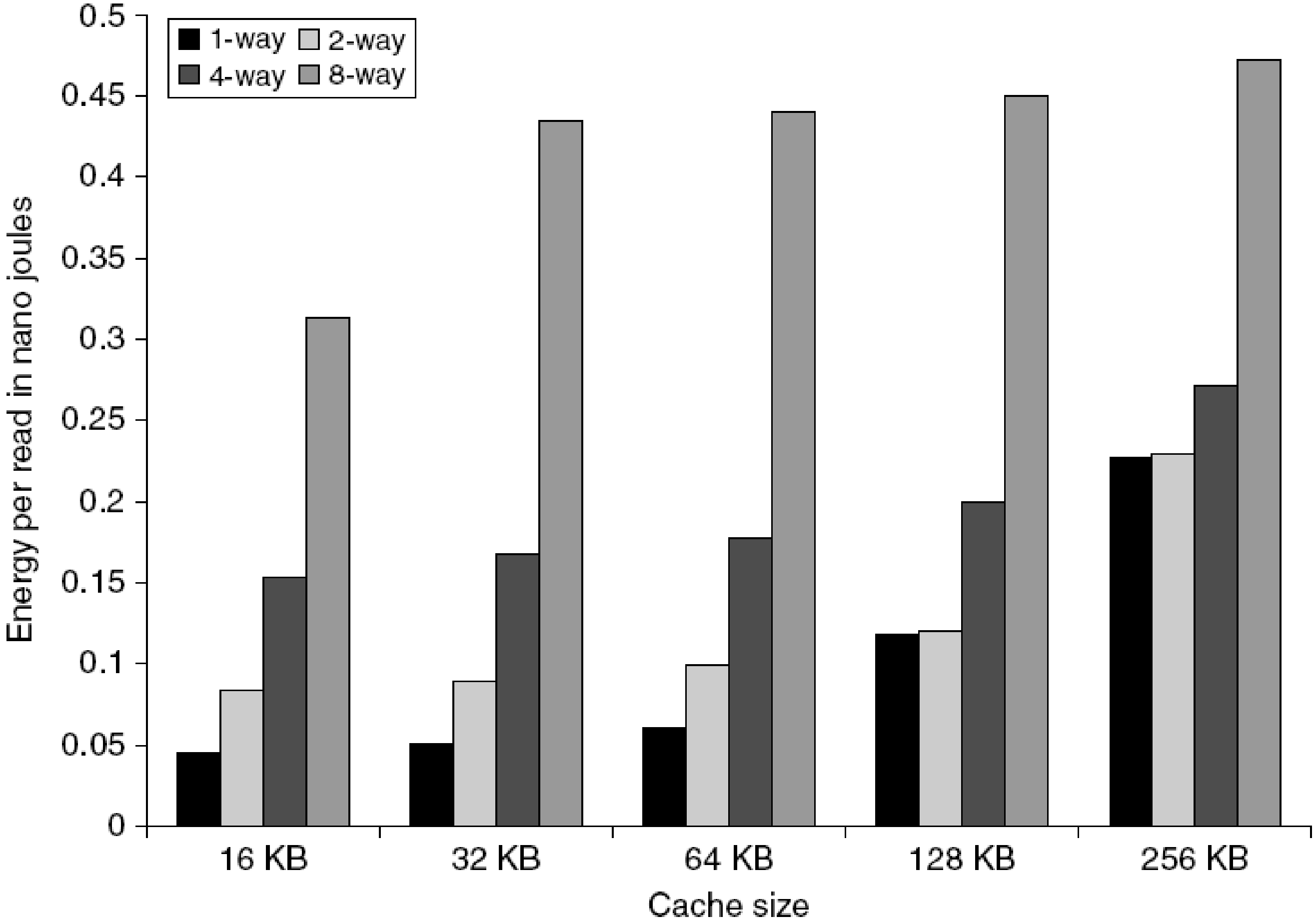
L1 Size and Associativity

Advanced Optimizations



Access time vs. size and associativity

L1 Size and Associativity



Energy per read vs. size and associativity

Way Prediction

- To improve hit time, predict the way to pre-set mux
  - Mis-prediction gives longer hit time
  - Prediction accuracy
    - > 90% for two-way
    - > 80% for four-way
    - I-cache has better accuracy than D-cache
  - First used on MIPS R10000 in mid-90s
  - Used on ARM Cortex-A8
- Extend to predict block as well
  - “Way selection”
  - Increases mis-prediction penalty

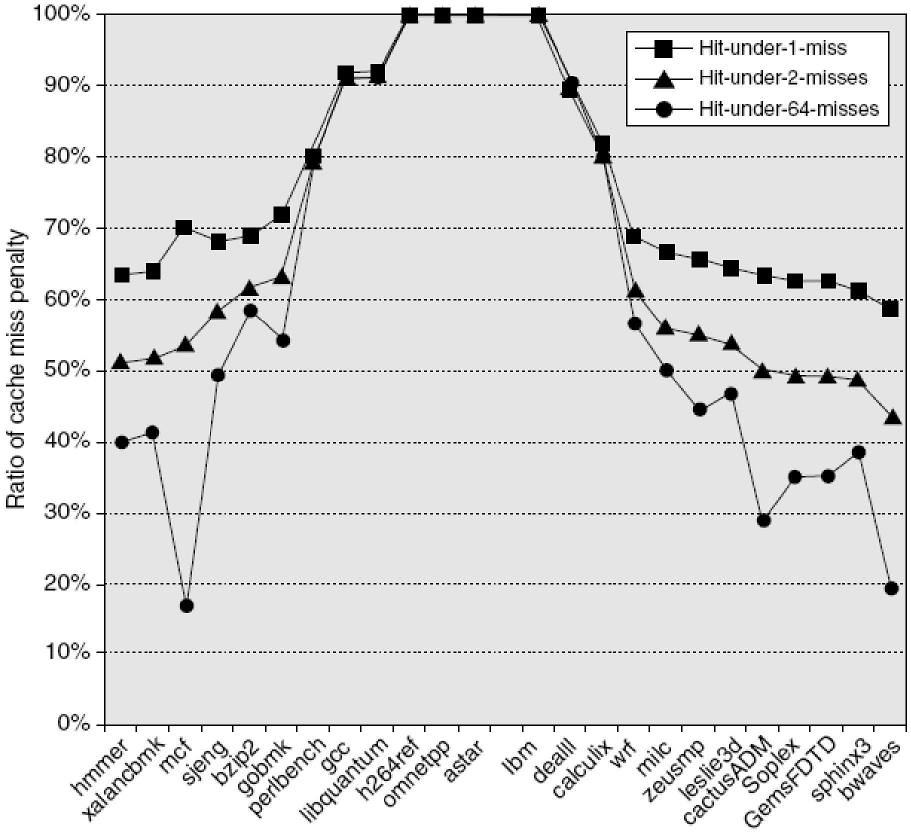
Pipelining Cache

- Pipeline cache access to improve bandwidth
  - Examples:
    - Pentium: 1 cycle
    - Pentium Pro – Pentium III: 2 cycles
    - Pentium 4 – Core i7: 4 cycles

- Increases branch mis-prediction penalty
- Makes it easier to increase associativity

Nonblocking Caches

- Allow hits before previous misses complete
  - "Hit under miss"
  - "Hit under multiple miss"
- L2 must support this
- In general, processors can hide L1 miss penalty but not L2 miss penalty



Multibanked Caches

- Organize cache as independent banks to support simultaneous access
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2

- Interleave banks according to block address

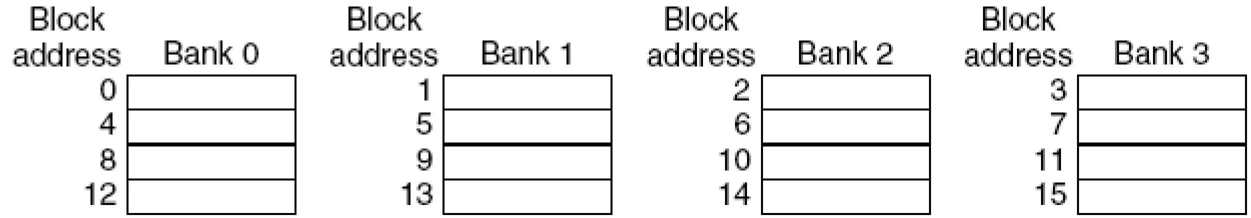


Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

Critical Word First, Early Restart

- Critical word first
  - Request missed word from memory first
  - Send it to the processor as soon as it arrives
- Early restart
  - Request words in normal order
  - Send missed work to the processor as soon as it arrives
- Effectiveness of these strategies depends on block size and likelihood of another access to the portion of the block that has not yet been fetched

Merging Write Buffer

- When storing to a block that is already pending in the write buffer, update write buffer
- Reduces stalls due to full write buffer
- Do not apply to I/O addresses

Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

No write buffering

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Write buffering

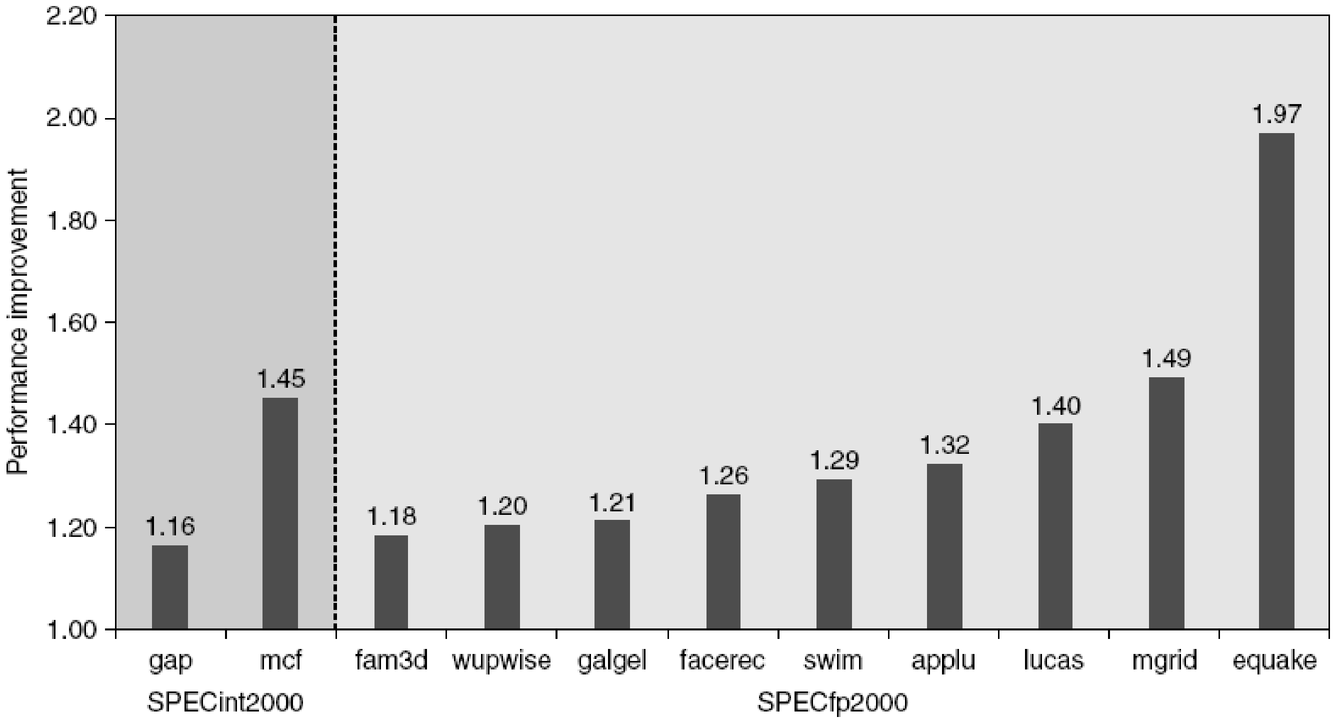
Compiler Optimizations

- Loop Interchange
  - Swap nested loops to access memory in sequential order

- Blocking
  - Instead of accessing entire rows or columns, subdivide matrices into blocks
  - Requires more memory accesses but improves locality of accesses

Hardware Prefetching

- Fetch two blocks on miss (include next sequential block)



Pentium 4 Pre-fetching

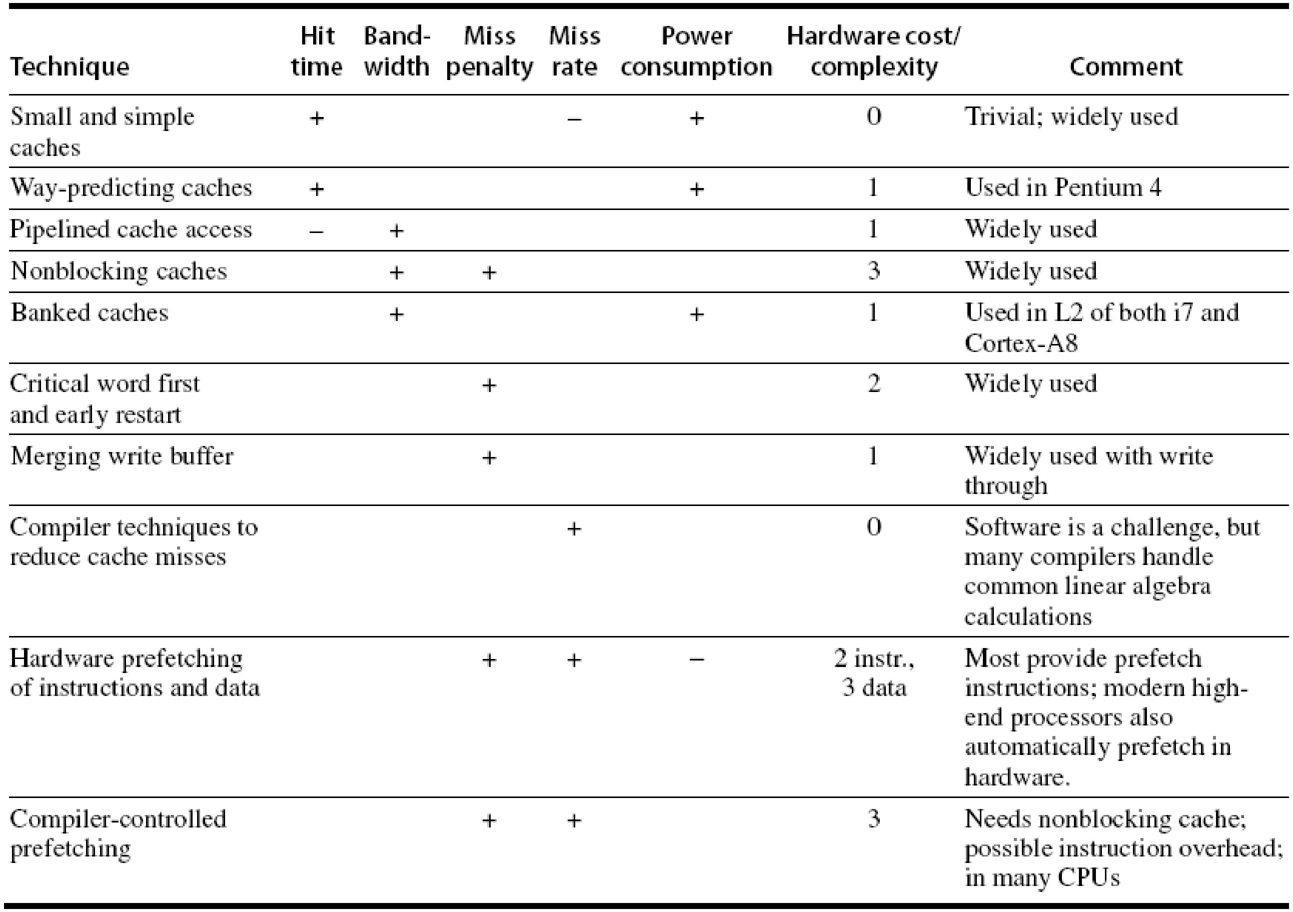
Compiler Prefetching

- Insert prefetch instructions before data is needed
- Non-faulting: prefetch doesn’t cause exceptions

- Register prefetch
  - Loads data into register
- Cache prefetch
  - Loads data into cache

- Combine with loop unrolling and software pipelining

Summary



Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.