Computer Organization and Architecture

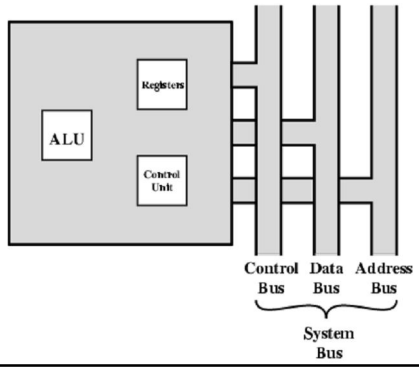
Chapter 12

CPU Structure and Function

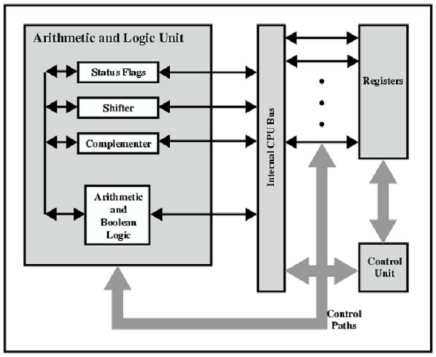
CPU Structure

- CPU must:
  - Fetch instructions
  - Interpret instructions
  - Fetch data
  - Process data
  - Write data
- These functions require
  - internal temporary storage
  - remembering location of instruction to fetch next

Simplified view of CPU With System Bus



More Detailed CPU Internal Structure



Register Organization

- CPU must have some working space (temporary storage); called "registers"
- Number and function vary between processor designs
  - One of the major design decisions
  - Top level of memory hierarchy
- Two main roles
  1. User Visible Registers
  2. Control and Status Registers

User Visible Registers

- A user visible register is simply a register than can referenced with the machine language of the processor
- Four categories
  - General Purpose
  - Data
  - Address
  - Condition Codes

General Purpose Registers (1)

- May be true general purpose (all registers are the same)
  - Orthogonal to the instruction set: any register can hold any operand for any instruction (clearly not the case with X86!)
  - May be restricted (each register has special functions)
- In some machines GP registers can be used for data or addressing
- In other machines there is a separation:
  - Data
    - Accumulator and count registers
  - Addressing
    - Segment, index, autoindex registers
    - Stack pointer
- Even on machines with true general purpose registers, if there is user-visible stack addressing then the stack pointer is special-purpose

General Purpose Registers (2)

- Design Decision:
  - Make them general purpose
    - Increase flexibility and programmer options
    - Increase instruction size & complexity
  - Make them specialized
    - Smaller (faster) instructions because of implied operands
    - Less flexibility

How Many GP Registers?

- Between 8 - 32
- Fewer = more memory references
- More registers do not reduce memory references and but they do take up processor real estate
- See also RISC – hundreds of registers in the machine (but only a few in use at any given moment)

How big?

- Large enough to hold full address
  - This was Intel's engineering kludge: 8086 index registers CANNOT hold a full address
- Large enough to hold full word
- Sometimes possible to combine two data registers for double-precision values

Condition Code (Flag) Registers

- Typically partially user-visible
- Sets of individual bits
  - e.g. result of last operation was zero
- Can be read (implicitly) by programs
  - e.g. Jump if zero
- Cannot (usually) be set directly (addressed) by programs
  - X86 has direct ops for carry flag only
  - STC, CLC, CMC
  - BT (Bit Test) Instruction

Machines without Condition Code Regs

- IA-64 (Itanium) and MIPS processors do not use condition code registers
- Conditional branch instructions specify a comparison and act on the result without storing the result in condition codes

Advantages and Disadvantages

Advantages	Disadvantages
1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed. 2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST AND BRANCH. 3. Condition codes facilitate multiway branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero.	1. Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the microprogrammer and compiler writer. 2. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections. 3. Often condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control, and atomic semaphore operations. 4. In a pipelined implementation, condition codes require special synchronization to avoid conflicts.

Control & Status Registers

- Four registers essential to instruction execution:
  - Program Counter (PC)
    - Address of next instruction to be fetched
  - Instruction Register (IR)
    - Current instruction fetched from mem
  - Memory Address Register (MAR)
    - Pointer to mem
  - Memory Buffer Register (MBR)
    - Word of data read or to be written
- Not all processors explicitly have MAR/MBR but equivalent functionality is present in all.

MAR/MBR

- A "staging area" for memory access
  - MAR connects to address bus
  - MBR connects to data bus
  - User registers exchange data with MBR
- Processor updates PC after instruction fetch; branch or skip can also update PC
- Fetched instruction is loaded into IR for decoding
- Data are exchanged with memory using MAR/MBR
- User-visible regs exchange data with MBR

ALU

- ALU typically has direct access to MBR and user registers
- Additional buffering registers typically are present at the "boundary"
- Serve as input and output registers for the ALU; exchange data with MBR and user-visible regs

Program Status Word

- Present in many processors –a set of bits indicating machine status (including condition codes)
- Typical:
  - Sign of last result
  - Zero
  - Carry
  - Equal
  - Overflow
  - Interrupt enable/disable
  - Supervisor mode (enable privileged instructions)
- X86 has flags/eflags plus control registers
- Instruction set has LMSW / SMSW as vestiges of 80286 PSW register

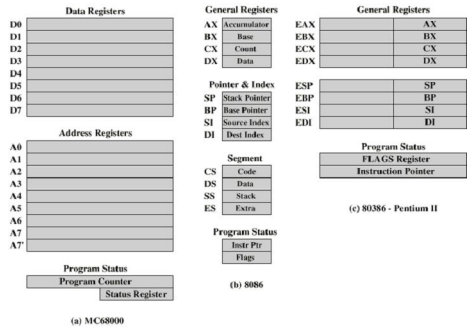
Supervisor Mode

- AKA Kernel mode, or with x86, Ring 0
  - Allows privileged instructions to execute
  - Used by operating system
  - Not available to user programs
  - Most control registers are only available in supervisor mode

Other status and control registers

- These vary by machine. Examples:
  - Pointer to current process information (x86 TR = Task Register)
  - Interrupt vector pointer (x86 IDTR)
  - System stack pointer (x86 SS:esp)
  - Page table pointer (x86 CR3)
  - I/O registers (not used in Intel x86)
- CPU design and operating system design are closely linked

Ex: Microprocessor Register Organizations



Intel and Motorola Differences

- Motorola
  - Uniform register sets (8 data, 9 address)
  - Two address registers used as stack pointers (supervisor and user mode)
  - 8, 16, 32 bit data in any of D0 – D7
  - No segmentation
- Intel
  - Every register is special purpose, some more than others
  - 8 and 16 bit data in AX, BX, CX, DX only
  - Segmentation needed for full 20-bit address
  - Many dedicated registers and implicit operands
  - Variable length machine language is very compact because of register design

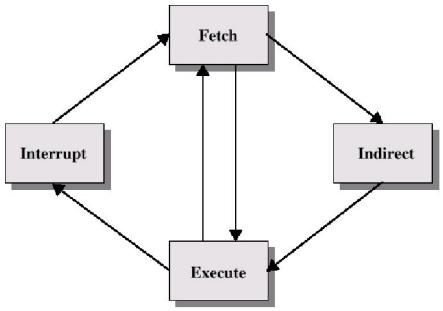
Instruction Cycle

- Chapter 3 revisited and elaborated

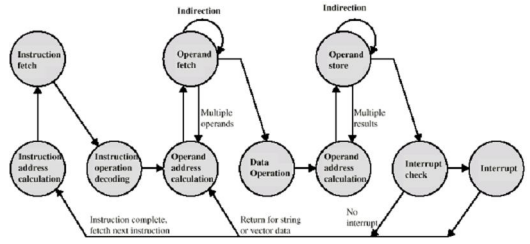
The Indirect Cycle

- Instruction execution may involve one or more memory operands/accesses
- If indirect addressing is used additional accesses are needed
- Can be thought of as additional instruction subcycle

Instruction Cycle with Indirect Cycle



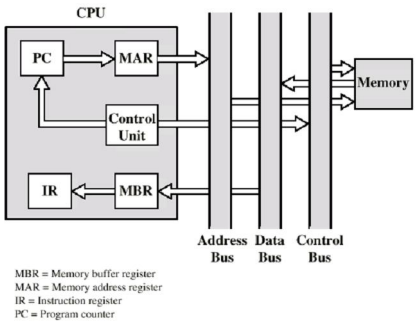
Instruction Cycle State Diagram



Data Flow (Instruction Fetch)

- Specifics depend on CPU design
- But In general:
- Fetch
  - PC contains address of next instruction
  - Address moved to MAR
  - Address placed on address bus
  - Control unit requests memory read
  - Result placed on data bus, copied to MBR, then to IR
  - Meanwhile PC incremented by 1 (or length of current instruction)

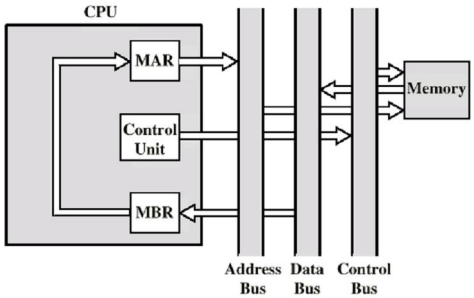
Data Flow (Fetch Diagram)



Data Flow (Data Fetch)

- IR is examined
- If indirect addressing, indirect cycle is performed
  - Right most N bits of MBR contain the address which is transferred to MAR
  - Control unit requests memory read
  - Result (address of operand) moved to MBR

Data Flow (Indirect Diagram)



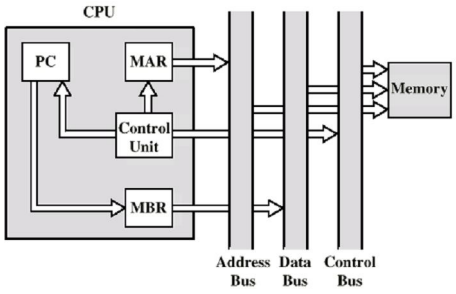
Data Flow (Execute)

- Fetch and indirect cycles are fairly simple and predictable; execution may take many forms
- Depends on instruction being executed
- May include
  - Memory read/write
  - Input/Output
  - Register transfers
  - ALU operations

Data Flow (Interrupt)

- Like fetch, simple and predictable
- Current PC saved to allow resumption after interrupt
- Contents of PC copied to MBR
- Special memory location (e.g. stack pointer) loaded to MAR
- MBR written to memory
- PC loaded with address of interrupt handling routine
- Next instruction (first of interrupt handler) can be fetched

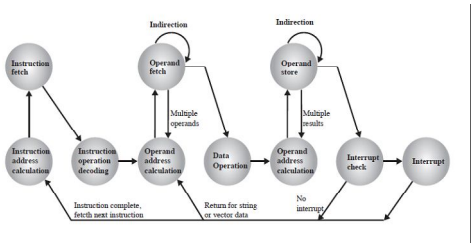
Data Flow (Interrupt Diagram)



Pipelining

- In pipelining we divide instruction execution into a number of stages
  - After one stage has completed, instruction moves down the pipeline to the next stage while the next instruction is started
  - Similar to a factory assembly line – we don't have to wait for a product to exit the line before starting to assemble another
- Simplified instruction cycle in Fig 12.5 has 10 stages
  - Intel Pentium 4: 20 stages
  - Pentium D: 31 stages

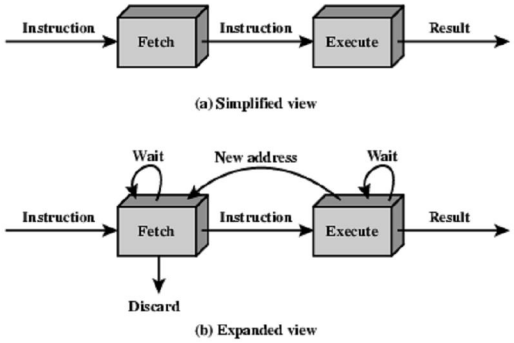
Simplified Instruction Cycle



Pipelining: Prefetch

- Instruction prefetch is the simplest form of pipelining
- Divide instruction into two phases: fetch and execute
  - Fetch accesses main memory
  - Execution usually does not access main memory
  - Instruction prefetch: fetch next instruction during execution of current instruction
  - Even 8088/8086 processors had small prefetch queues (4 and 6 bytes) to allow multiple instructions to be fetched

Two Stage Instruction Pipeline



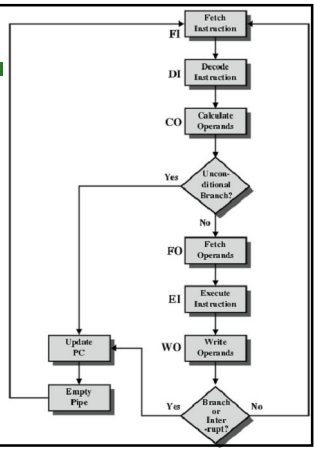
Improved Performance

- But not doubled:
  - Fetch is usually shorter than execution
    - Prefetch more than one instruction?
  - Any jump or branch means that prefetched instructions are not the required instructions
- So add more stages to improve performance

Pipelining

- A Six stage pipeline:
  1. FI Fetch instruction
  2. DI Decode instruction
  3. CO Calculate operands (i.e. EAs)
  4. FO Fetch operands
  5. EI Execute instructions
  6. WO Write operand
- Overlap these operations so that while instruction 1 is being decoded, instruction 2 is being fetched etc.
- Not all instructions use all six stages

Six Stage Instruction Pipeline



Timing Diagram for Instruction Pipeline Operation

Time

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

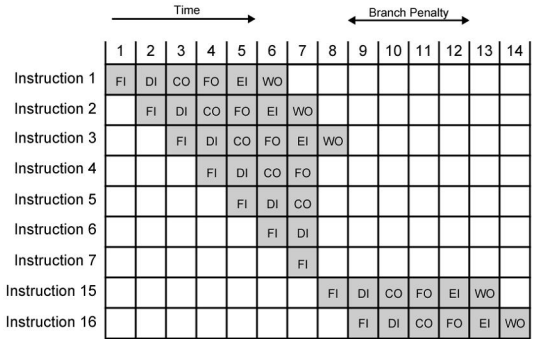
Assumptions

- Timing diagram shows 9 instructions in 14 clocks rather than 54
- Each operation same length of time
- No memory conflicts
  - Values may be in cache
  - FO, WO may be null

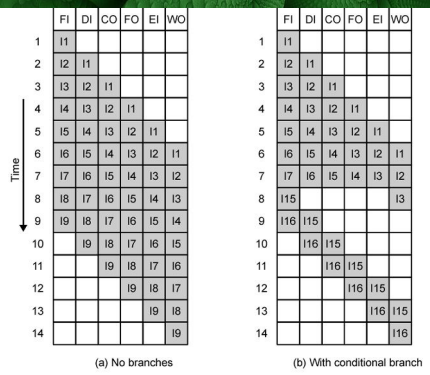
Complications

- Pipeline speed is limited to speed of slowest stage
- Conditional branches: result not known until WO stage
- CO stage may require output from previous instruction

The Effect of a Conditional Branch on Instruction Pipeline Operation



Alternative Pipeline Depiction

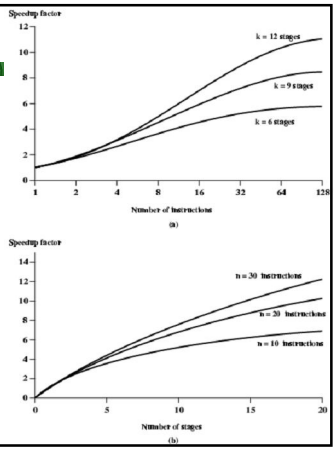


Number of stages

- Appears that more pipeline stages => better performance
- But:
  - Pipeline overhead: moving from buffer to buffer, performing prep and delivery functions can take time
  - Sequential data dependencies slow down the pipeline
  - Amount of control logic increases: logic controlling gating between stages can be more complex than the stages being controlled

Speedup Factors withPipelining

- No branches encountered
- Speedup factor = kn / (k + n – 1)
- Where k = # stages and n = # instructions
- As n -> ∞ speedup approaches k

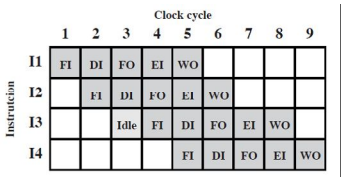


Pipeline Hazards

- Hazards are situations where a portion of the pipeline has to stall or wait because continued execution is not possible
- Resource Hazards
  - Two or more instructions in the pipeline need the same resource (e.g., multiply unit)
- Data Hazards
  - Conflict in accessing an operand location (e.g., a register)
- Control Hazards (Branch Hazards)
  - Pipeline makes wrong decision about branch prediction and has to discard instructions

Resource Hazards

- Example: single port memory, no cache. Instruction fetch and data read cannot be performed in parallel
- Here the Fetch Operand (FO) and Fetch Instruction (FI) stages contend for the same resource (memory)



Resolving Resource Hazards

- Aside from memory other resources that might be points of contention are registers, ALU, or ALU components (shifter, multiply/divide unit, etc.)
- One resolution is to reduce contention by adding more units or access points (ports into memory)
- Another approach is to use a reservation table (see Appendix I – online)

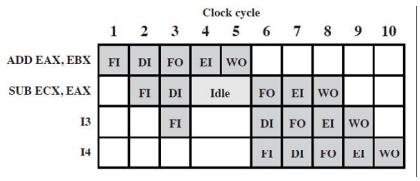
Data Hazards

- Conflict in access of an operand location
  - Two instructions in sequence access the a particular mem or reg operand
  - If executed in strict sequence there is no problem
  - But in a pipeline it is possible for operands to be updated in a different order than strict sequence producing different result

Data Hazard Example

```
add eax, ebx    ; eax <- eax+ebx
sub ecx, eax    ; ecx <- ecx-eax
```

- The ADD does not update eax until stage 5 (Write Operand) at clock 5. But SUB needs operand at stage 3 (FO) at clock 4. Pipeline stalls for 2 clocks



Types of Data Hazard

- Read-after-Write or "true dependency" (RAW)
  - Instruction modifies operand and a later instruction reads the same operand. Read cannot take place before the write. (Previous example)
- Write-after-Read or "antidepency" (WAR)
  - Instruction reads an operand and a later instruction writes the same operand. Write cannot take place before the read.
- Write-after-Write or "output depency" (WAW)
  - Two instructions writes to the same operand. Write operations must execute in correct order.

A note on terminology

- Everybody agrees that this instruction sequence has a data hazard, and that it is a "true data dependency"

```
ADD eax, ecx
MOV ebx, eax
```

- Unfortunately, in the literature some people describe this as "read after write" (RAW) while others describe it as "write after read" (WAR)
  - The RAW description describes the instruction sequence as it appears in the instruction stream and as it should be correctly executed by the processor. The Read MUST take place after the Write
  - The WAR description describes the hazard, i.e., it describes the incorrect execution sequence where the Write actually occurs after the read, so the result is not correct
- The textbook uses RAW in Ch. 12 and WAR in Ch. 14.
- We will use the RAW approach (describe the instruction stream as it should be executed)

Control Hazards: Dealing with Branches

- Branches are the primary impediment to ensuring optimal pipeline efficiency
- Several approaches to dealing with conditional branches
  - Multiple Streams
  - Prefetch Branch Target
  - Loop buffer
  - Branch prediction
  - Delayed branching

Multiple Streams

- Brute force approach: Have two pipelines and prefetch each branch into a separate pipeline
- Then use the appropriate pipeline
- Drawbacks:
  - Leads to bus & register contention
  - Multiple branches lead to further pipelines being needed
- Used in IBM 370/168 and 3033

Prefetch Branch Target

- Target of branch is prefetched in addition to instructions following branch
- Keep target until branch is executed
- Used by IBM 360/91
- Only gains one instruction in pipeline if branch is taken

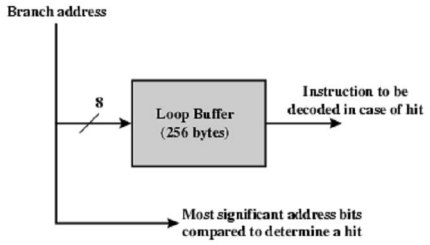
Loop Buffer

- A small, very fast cache memory contains n most recently fetched instructions in sequence
  - Maintained by fetch stage of pipeline
  - Check buffer for branch target before fetching from memory
- Very good for small loops or jumps
- Behaves as a small instruction cache
  - Contains only instructions fetched in sequence
  - Smaller and cheaper than associative cache
  - If branch target is near the branch then it will already be in the loop buffer

Loop Buffer

- Difference between loop buffer and instruction cache is that the loop buffer does not use associative memory – instructions are in sequence
- Used by CDC, CRAY-1, Motorola 68010 for DBcc (decrement and branch on condition) instruction only; 3 words

Loop Buffer Diagram



Branch Prediction

- Basically guessing whether or not a branch will be taken: Several strategies can be used:
  - Predict always taken
  - Predict never taken
  - Predict by opcode
  - Taken/Not Taken switch
  - Branch history table
- First two strategies are static (no dependence on instruction history); last two are dynamic and vary with instruction history
- Predict by opcode is a static strategy that varies by opcode.

Branch Prediction: Never/Always

- Predict never taken/always taken
- Simple approach
  - Assume that jump will not / will happen
  - Always fetch next instruction / branch target
  - 68020 & VAX 11/780 use never taken
  - VAX will not prefetch after branch if a page fault would result (O/S v CPU design)
- Studies suggests > 50% of instructions take branch
- But probability of page fault is higher with the branch

Branch Prediction by Opcode

- Some instructions are more likely to result in a jump than others
  - Example: Intel LOOP instructions
  - Can get up to 75% success
- Variation: predict that backwards branches taken, forward are not taken

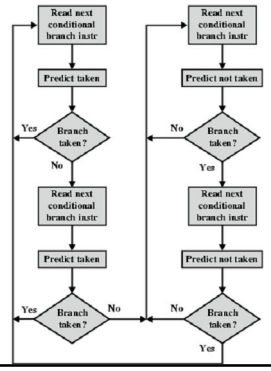
Branch Prediction: History

- One or more Taken/Not taken bits are associated with each conditional branch
- Bits are stored in high speed memory; either associated with instruction in cache or maintained in a small table
- With a single bit we can only record taken or not taken on last execution
- For loops with many iterations this will miss twice (once at start, once at end)

Branch Prediction: 2 bits

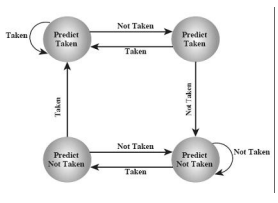
- With two bits we can either record the result of the last two executions or some other state indicator
- Typical approach in Fig 12.16:
  - First predict taken
  - Continue to predict taken until two successive predictions are wrong
  - Then predict not taken
  - Continue to predict not taken until two successive predictions are wrong
- 2-bit prediction usually has only one miss per loop

Branch Prediction Flowchart



Branch Prediction: FSM

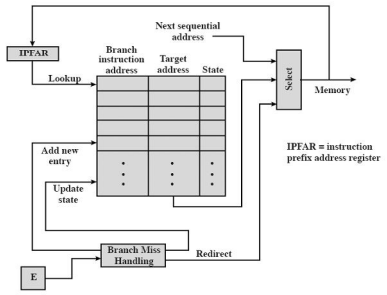
- A more compact way to express this branch prediction scheme is a finite state machine
- Start in upper left hand corner



Branch History Table

- If decision is made to take the branch the target instruction cannot be fetched until the target address operand is decoded
- Branch history table is a small cache memory associated with the fetch stage of a pipeline
- Each entry has three elements:
  - Instruction address
  - Branch history bits
  - Target address (or instruction)

Branch History Table



Design issues

- Issue is size of branch table
- For large tables, can be up to 94% correct
- Indexed by LS bits of instruction address
- Many branches in small block of code can reduce accuracy

N-bit branch history prediction

- Use saturating counter (no wrap around)
- Keep an n-bit saturating counter for each branch.
- Increment it on branch taken and decrement it on branch not taken .
- If the counter is greater than or equal to half its maximum value, predict the branch as taken.
- This can be done for any n
- n=2 performs almost as good as other values for n.

Delayed Branch

- Improve pipeline performance by rearranging instructions so that branch instructions execute after they actually appear in the code
- Use in some RISC machines
- Discussed in more detail later

Pentium Branch Prediction

- The previous discussion is simplified
- See

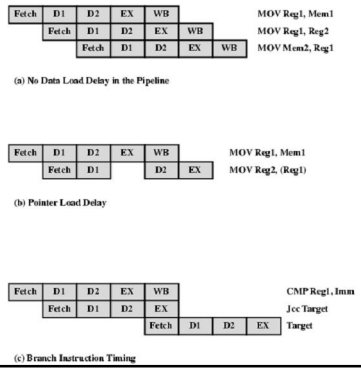
http://www.x86.org/articles/branch/branchprediction.htm

- Interesting discussion of the original not-very-good branch prediction in Pentium
- Later processors use a 2-level branch prediction mechanism that can correctly predict repetitive patterns

Intel 80486 Pipelining (5 stages)

- Fetch
  - From cache or external memory
  - Put in one of two 16-byte prefetch buffers
  - Fill buffer with new data as soon as old data consumed
  - Average 5 instructions fetched per load
  - Independent of other stages to keep buffers full
- Decode stage 1 gets 3 bytes from fetch buffers
  - Opcode & address-mode info
  - This info found in at most first 3 bytes of instruction
  - Can direct D2 stage to get rest of instruction
- Decode stage 2
  - Expand opcode into control signals
  - Computation of complex address modes
- Execute
  - ALU operations, cache access, register update
- Writeback
  - Update registers & flags
  - Results sent to cache & bus interface write buffers

80486 Instruction Pipeline Examples



X86 Integer Unit Registers

- Includes models

(a) Integer Unit in 32-bit Mode

Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

(b) Integer Unit in 64-bit Mode

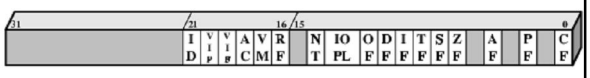
Type	Number	Length (bits)	Purpose
General	16	32	General-purpose user registers
Segment	6	16	Contain segment selectors
RFLAGS	1	64	Status and control bits
Instruction Pointer	1	64	Instruction pointer

Floating Point Unit

Type	Number	Length (bits)	Purpose
Numeric	8	80	Hold floating-point numbers
Control	1	16	Control bits
Status	1	16	Status bits
Tag Word	1	16	Specifies contents of numeric registers
Instruction Pointer	1	48	Points to instruction interrupted by exception
Data Pointer	1	48	Points to operand interrupted by exception

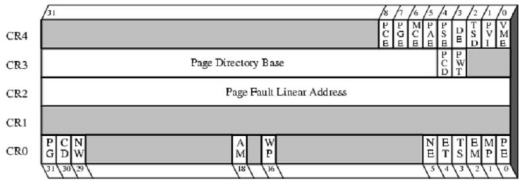
EFLAGS Register

- Note that upper 32 bits of 64 bit rflags are unused



ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag
DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

Control Registers

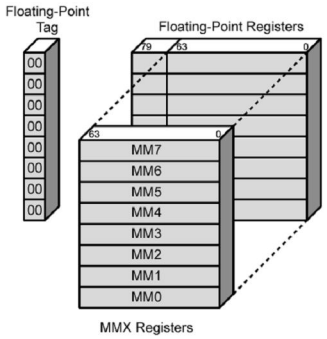


PCE = Performance Counter Enable
PGE = Page Global Enable
MCE = Machine Check Enable
PAE = Physical Address Extension
PSE = Page Size Extensions
DE = Debug Extensions
TSD = Time Stamp Disable
PVI = Protected Mode Virtual Interrupt
VME = Virtual 8086 Mode Extensions
PCD = Page-level Cache Disable
PWT = Page-level Writes Transparent
PG = Paging
CD = Cache Disable
NW = Not Write Through
AM = Alignment Mask
WP = Write Protect
NE = Numeric Error
ET = Extension Type
TS = Task Switched
EM = Emulation
MP = Monitor Coprocessor
PE = Protection Enable

MMX Register Mapping

- MMX uses several 64 bit data types
- Use 3 bit register address fields
  - 8 registers
- No MMX specific registers
  - Aliasing to lower 64 bits of existing floating point registers
  - Direct addressing instead of stack addressing
  - Upper 16 bits set to all 1's (NaN) so that register is not valid for subsequent FP operation
- EMMS (Empty MMX State) instruction used at end of MMX block

Mapping of MMX Registers to Floating-Point Registers



Pentium Interrupt Processing

- Interrupts (signal from hardware)
  - Maskable
  - Nonmaskable
- Exceptions (generated from software)
  - Processor detected (integer div 0, page fault)
  - Programmed (INTO, INT, INt3, BOUND)
- 5 priority classes provide predictable service order

Real Mode interrupt processing

- In real mode (16-bit code) interrupt vectors are stored in the first physical 1KB of memory
- Addresses 0000:0000 through 0000:03FF
- Interrupt vectors are 32-bit segment:offset addresses
- 256 * 4 bytes occupies first 1KB of memory

Protected Mode Interrupt Processing

- Interrupts are vectored through the Interrupt Descriptor Table (IDT)
- The IDTR register points to base of IDT
- 8-byte interrupt descriptor is very similar to a segment descriptor in the LDT or GDT

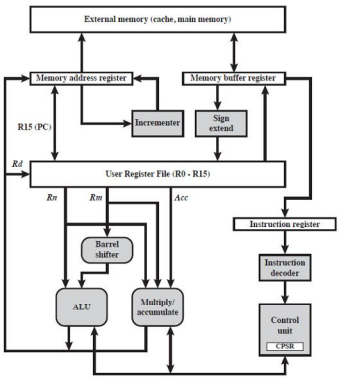
The ARM Processor

- Key attributes:
  - Moderate array of uniform registers; more than CISC machines but fewer than many RISC machines
  - Load/store model of data processing
  - Uniform fixed length instruction set (32 bits standard and 16 bits Thumb)
  - Shift or rotate processing with all data/logical instructions. Separate ALU and shifter units
  - Small number of addressing modes (but still somewhat richer than most RISC machines)
  - Auto-increment and auto-decrement addressing modes
  - Conditional execution of instructions removes many branch processing issues

ARM Processor Organization

- ARM organization varies significantly from one implementation to the next and from one version to the next
- We will discuss a generalized and simplified model
- Note that ARM instructions use 3-operand addresses: 2 source registers and a destination
- Output of ALU can go into a register or can be used to generate a memory address in the MAR

Simplified Organization



Processor Modes

- Early microprocessors had only one operating mode; everything was accessible to the current program
- Many operating systems use only two processor modes: user and kernel (or supervisor)
- The ARM processor has 7 operating modes
- Most applications execute in User mode.
  - Program cannot access protected system resources or change mode except by causing an exception

Privileged Modes

- The remaining six modes are used to run system software
- Reasons for large number of modes are
  - OS can tailor systems software to a variety of circumstances
  - Certain registers are dedicated for use each of the privileged modes, allowing fast context switching
- Five modes are known as exception modes
  - Entered when specific exceptions occur
  - Full access to system resources
  - Can switch modes
  - Dedicated registers for each mode substitute for some of the user mode registers

Exception Modes

- Supervisor mode: normal running mode for OS
  - Entered when processor encounters a software interrupt instruction (standard way to invoke system services)
- Abort mode: entered in response to memory faults
- Undefined Mode: Entered when execution of an unsupported instruction is attempted
- Fast interrupt mode: Entered when CPU receives signal from the designated fast interrupt source
  - Fast interrupts cannot be interrupted but can interrupt a normal interrupt
- Interrupt Mode: for all but the fast interrupt.
  - Can only be interrupted by a fast interrupt

System Mode

- Not entered by any exception
- Uses same registers as user mode
- Used for operating system tasks; can be interrupted by any of the five exception categories

Register Organization

- Total of 37 32-bit registers
  - 31 are referred to as general purpose register although some such R15 (PC) have special purposes
  - Six program status registers
- Registers are arranged in partially overlapping banks.Each processing mode has a bank.
- At any time sixteen numbered registers and one or two program status registers are visible for a total of 16 or 17 visible registers
  - R0 through R7 plus R15 (PC) and Current Program Status Register (CPSR) are shared by all modes
  - R8 through R12 are shared by all modes except Fast Interrupt Mode which has R8_fiq .. R12_fiq
  - All exception modes have their own copies of R13 and R14
  - All exception modes have a dedicated Saved Program Status Register (SPSR)

General Purpose Registers

- R0 through R12 are general purpose registers
  - R13 is the stack pointer; each mode has its own stack
  - R14 is the link register, used to store function and exception return addresses
  - R15 is the program counter

ARM Registers

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13 (SP)	R13 (SP)	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14 (LR)	R14 (LR)	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

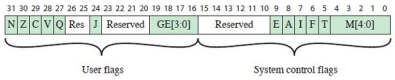
Shading indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode.

SP = stack pointer
LR = link register
PC = program counter

CPSR = current program status register
SPSR = saved program status register

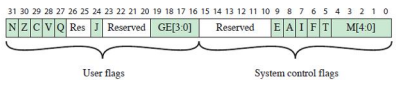
Program Status Register

- The CPSR is accessible in all modes.
- CPSR is preserved in SPSR in exception modes
- Upper 16 bits contain user flags
  - Condition code flags N,Z,C,V
  - Q flag indicates overflow or saturation in SIMD instructions
  - J bit related to Jazelle instructions (allow Java bytecode to execute directly on the ARM)
  - GE bits are set by SIMD instructions (greater than or equal) for each operand



System Control Flags

- E bit control endianness for data (ignored for code)
- A,I,F are interrupt disable bits.
  - A disables imprecise operand data aborts
  - I disables IRQ interrupts
  - F disables FIQ
- T bit controls normal / Thumb interpretation of code
- Mode bits indicate processor mode



Interrupt Processing

- ARM supports seven types of exception.
  - Vectors for each type are stored at fixed addresses in low memory 0x00000000 – 0x0000001F (vector at address 0x00000014 is reserved)
  - Multiple interrupts are handled in priority order (0 highest)
- Exception handling:
  - Finish current instruction
  - Save state of CPSR in SPSR register for exception mode
  - PC (R15) is saved in Link Register (R14) for exception mode
  - Return by copying SPSR back to CPSR and R14 to R15

Interrupt Vectors

Exception type	Mode	Normal entry address	Description
Reset	Supervisor	0x00000000	Occurs when the system is initialized.
Data abort	Abort	0x00000010	Occurs when an invalid memory address has been accessed, such as if there is no physical memory for an address or the correct access permission is lacking.
FIQ (fast interrupt)	FIQ	0x0000001C	Occurs when an external device asserts the FIQ pin on the processor. An interrupt cannot be interrupted except by an FIQ. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching. A fast interrupt cannot be interrupted.
IRQ (interrupt)	IRQ	0x00000018	Occurs when an external device asserts the IRQ pin on the processor. An interrupt cannot be interrupted except by an FIQ.
Prefetch abort	Abort	0x0000000C	Occurs when an attempt to fetch an instruction results in a memory fault. The exception is raised when the instruction enters the execute stage of the pipeline.
Undefined instructions	Undefined	0x00000004	Occurs when an instruction not in the instruction set reaches the execute stage of the pipeline.
Software interrupt	Supervisor	0x00000008	Generally used to allow user mode programs to call the OS. The user program executes a SWI instruction with an argument that identifies the function the user wishes to perform.