Digital Registers

Twanna Christian

First Edition, 2012

ISBN 978-81-323-3430-9

Published by:
Research World
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1 - Control Register

Chapter 2 - Hardware Register and Processor Register

Chapter 3 - Flag Word and FLAGS Register (Computing)

Chapter 4 - Linear Feedback Shift Register

Chapter 5 - Program Counter and Register File

Chapter 6 - Register Renaming

Chapter 7 - Register Window and Shift Register

Chapter 8 - Word (Computing)

Chapter 9 - Byte

Chapter 10 - 64-bit

Chapter 11 - 16-bit, 26-bit and 31-bit

Chapter 12 - 32-bit, 36-bit, 8-bit

Chapter-1

Control Register

A **control register** is a processor register which changes or controls the general behavior of a CPU or other digital device. Common tasks performed by control registers include interrupt control, switching the addressing mode, paging control, and coprocessor control.

Control registers in x86 series

CR0

The CR0 register is 32 bits long on the 386 and higher processors. On x86-64 processors in long mode, it (and the other control registers) are 64 bits long. CR0 has various control flags that modify the basic operation of the processor.

Bit	Name	Full Name	Description
31	PG	Paging	If 1, enable paging and use the CR3 register, else disable paging
30	CD	Cache disable	Globally enables/disable the memory cache
29	NW	Not-write through	Globally enables/disable write-back caching
18	AM	Alignment mask	Alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3
16	WP	Write protect	Determines whether the CPU can write to pages marked read-only
5	NE	Numeric error	Enable internal x87 floating point error reporting when set, else enables PC style x87 error detection
4	ET	Extension type	On the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387
3	TS	Task switched	Allows saving x87 task context only after x87 instruction used after task switch
2	EM	Emulation	If set, no x87 floating point unit present, if clear, x87 FPU present

1	MP	Monitor co-processor	Controls interaction of WAIT/FWAIT instructions with TS flag in CR0
0	PE	Protected Mode Enable	If 1, system is in protected mode, else system is in real mode

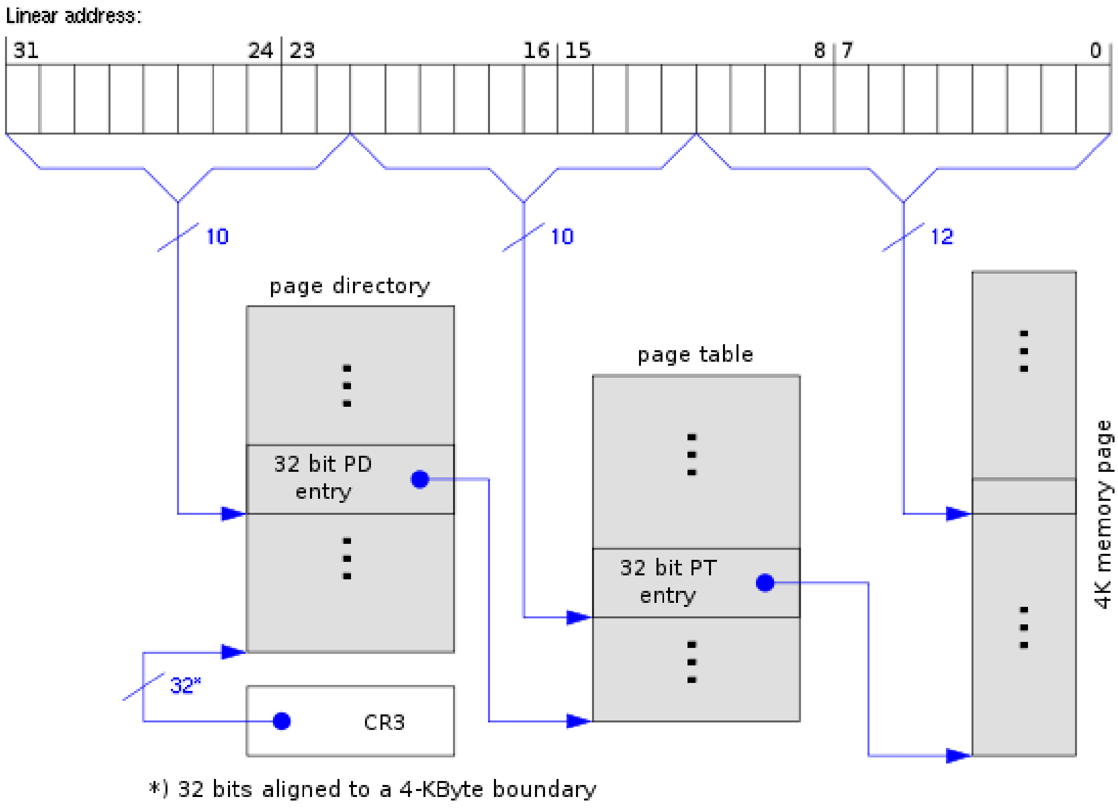
CR1

Reserved

CR2

Contains a value called Page Fault Linear Address (PFLA). When a page fault occurs, the address the program attempted to access is stored in the CR2 register.

CR3



Typical use of CR3 in address translation with 4 KiB pages.

Used when virtual addressing is enabled, hence when the PG bit is set in CR0. CR3 enables the processor to translate virtual addresses into physical addresses by locating the page directory and page tables for the current task. Typically, the upper 20 bits of CR3 become the *page directory base register* (PDBR).

CR4

Used in protected mode to control operations such as virtual-8086 support, enabling I/O breakpoints, page size extension and machine check exceptions.

Bit	Name	Full Name	Description
18	OSXSAVE	XSAVE and Processor Extended States Enable	
17	PCIDE	PCID Enable	If set, enables process-context identifiers (PCIDs).
14	SMXE	SMX Enable	
13	VMXE	VMX Enable	
10	OSXMMEXCPT	Operating System Support for Unmasked SIMD Floating-Point Exceptions	If set, enables unmasked SSE exceptions.
9	OSFXSR	Operating system support for FXSAVE and FXSTOR instructions	If set, enables SSE instructions and fast FPU save & restore
8	PCE	Performance-Monitoring Counter enable	If set, RDPMC can be executed at any privilege level, else RDPMC can only be used in ring 0.
7	PGE	Page Global Enabled	If set, address translations (PDE or PTE records) may be shared between address spaces.
6	MCE	Machine Check Exception	If set, enables machine check interrupts to occur.
5	PAE	Physical Address Extension	If set, changes page table layout to translate 32-bit virtual addresses into extended 36-bit physical addresses.
4	PSE	Page Size Extensions	If unset, page size is 4 KB, else page size is increased to 4 MB (ignored with PAE set).
3	DE	Debugging Extensions	
2	TSD	Time Stamp Disable	If set, RDTSC instruction can only be executed when in ring 0, otherwise RDTSC can be used at any privilege level.
1	PVI	Protected-mode Virtual Interrupts	If set, enables support for the virtual interrupt flag (VIF) in protected mode.

0	VME	Virtual 8086 Mode Extensions	If set, enables support for the virtual interrupt flag (VIF) in virtual-8086 mode.

Additional Control registers in x86-64 series

EFER

Extended Feature Enable Register (EFER) is a register added in the AMD K6 processor, to allow enabling the SYSCALL/SYSRET instruction, and later for entering and exiting long mode.

Bit	Purpose
63:12	Reserved
11	Execute-disable bit enable (NXE)
10	IA-32e mode active (LMA)
9	Reserved
8	IA-32e mode enable (LME)
7:1	Reserved
0	SysCall enable (SCE)

Chapter-2

Hardware Register and Processor Register

Hardware register

In digital electronics, especially computing, a **hardware register** stores bits of information, in a way that all the bits can be written to or read out simultaneously. The hardware registers inside a central processing unit (CPU) are called processor registers. Signals from a state machine to the register control when registers transmit to or accept information from other registers. Sometimes the state machine routes information from one register through a functional transform, such as an adder unit, and then to another register that stores the results.

Typical uses of hardware registers include *configuration* and start-up of certain features, especially during initialization, *buffer storage* e.g. video memory for graphics cards, input/output (I/O) of different kinds, and *status reporting* such as whether a certain event has occurred in the hardware unit.

Reading a hardware register in "peripheral units" -- computer hardware outside the CPU—involves accessing its memory-mapped I/O address or port-mapped I/O address with a "load" or "store" instruction, issued by the processor. Hardware registers are addressed in words, but sometimes only use a few bits of the word read in to, or written out to the register.

Strobe registers have the same interface as normal hardware registers, but instead of storing data, they trigger an action each time they are written to (or, in rare cases, read from). They are a means of signaling.

Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register". Registers can be implemented in a wide variety of ways, including register files, standard SRAM, individual flip-flops, or high speed core memory.

In addition to the "programmer-visible" registers that can be read and written with software, many chips have internal registers that are used for state machines and pipelining; for example, registered memory.

Commercial design tools such as Socrates Bitwise by Duolog Technologies, simplify and automate memory-mapped register specification and code generation for hardware, firmware, hardware verification, testing and documentation.,,

Using IP-XACT IEEE 1685, commercial design tools, such as MRV Magillem Register View by MAGILLEM, provide a real synchronization between the register description and the RTL hardware platform description, then collaborative work in the design flow can be addressed. This hardware registers alignment becomes critical when multiple levels of abstraction are used when switching from TLM to RTL during IP integration.

SPIRIT IP-XACT and DITA SIDSC XML define standard XML formats for memory-mapped registers.,,

Because write-only registers make debugging almost impossible, lead to the read-modify-write problem, and also make it unnecessarily difficult for the Advanced Configuration and Power Interface to determine the device's state when entering sleep mode in order to restore that state when exiting sleep mode, many programmers tell hardware designers to make sure that all writable registers are also readable. However, there are some cases when reading certain types of hardware registers is useless. For example, a strobe register bit that generates a one cycle pulse into specialized hardware will always read logic 0.

Processor register

In computer architecture, a **processor register** (or **general purpose register**) is a small amount of storage available on the CPU whose contents can be accessed more quickly than storage available elsewhere. Typically, this specialized storage is not considered part of the normal memory range for the machine. Most, but not all, modern computers adopt the so-called load-store architecture. Under this paradigm, data is loaded from some larger memory — be it cache or RAM — into registers, manipulated or tested in some way (using machine instructions for arithmetic/logic/comparison) and then stored back into memory, possibly at some different location. A common property of computer programs is locality of reference: the same values are often accessed repeatedly; and holding these frequently used values in registers improves program execution performance.

Processor registers are at the top of the memory hierarchy, and provide the fastest way for a CPU to access data. The term is often used to refer only to the group of registers that are directly encoded as part of an instruction, as defined by the instruction set. More properly, these are called the "architectural registers". For instance, the IA-32 instruction set defines a set of 32-bit registers, but a CPU that implements the x86 instruction set will often contain many more registers than just these registers.

Allocating frequently used variables to registers can be critical to a program's performance. This action (register allocation) is performed by a compiler in the code generation phase.

Categories of registers

Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register". A processor often contains several kinds of registers, that can be classified accordingly to their content or instructions that operate on them:

- **User-accessible Registers** - The most common division of user-accessible registers is into data registers and address registers.
- **Data registers** are used to hold numeric values such as integer and floating-point values. In some older and low end CPUs, a special data register, known as the accumulator, is used implicitly for many operations.
- **Address registers** hold addresses and are used by instructions that indirectly access memory.
    - Some processors contain registers that may only be used to hold an address or only to hold numeric values (in some cases used as an index register whose value is added as an offset from some address); others allow registers to hold either kind of quantity. A wide variety of possible addressing modes, used to specify the effective address of an operand, exist.
    - A stack pointer, sometimes called a stack register, is the name given to a register that can be used by some instructions to maintain a stack.
- **Conditional registers** hold truth values often used to determine whether some instruction should or should not be executed.
- **General purpose registers** (**GPR**s) can store both data and addresses, i.e., they are combined Data/Address registers.
- **Floating point registers** (**FPR**s) store floating point numbers in many architectures.
- **Constant registers** hold read-only values such as zero, one, or pi.
- **Vector registers** hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).
- **Special purpose registers (SPR)** hold program state; they usually include the program counter (aka instruction pointer), stack pointer, and status register (aka processor status word). In embedded microprocessors, they can also correspond to specialized hardware elements.
    - **Instruction registers** store the instruction currently being executed.
- In some architectures, **model-specific registers** (also called *machine-specific registers*) store data and settings related to the processor itself. Because their meanings are attached to the design of a specific processor, they cannot be expected to remain standard between processor generations.
- **Control and status registers** - It has three types. Program counter, instruction registers, Program status word (PSW).

- Registers related to fetching information from RAM, a collection of storage registers located on separate chips from the CPU (unlike most of the above, these are generally not *architectural* registers):
    - Memory buffer register
    - Memory data register
    - Memory address register
    - Memory Type Range Registers (MTRR)

Hardware registers are similar, but occur outside CPUs.

Some examples

Architecture	Integer registers	Double FP registers
x86	8	8
x86-64	16	16
IBM/360	16	4
Z/Architecture	16	16
Itanium	128	128
UltraSPARC	32	32
POWER	32	32
Alpha	32	32
6502	3	0
PIC microcontroller	1	0
AVR microcontroller	32	0
ARM	16	16

The table shows the number of registers of several mainstream architectures. Note that the stack pointer (ESP) is counted as an integer register on x86-compatible processors, even though there are a limited number of instructions that may be used to operate on its contents. Similar caveats apply to most architectures.

Register usage

The number of registers available on a processor and the operations that can be performed using those registers has a significant impact on the efficiency of code generated by optimizing compilers. The Strahler number defines the minimum number of registers required to evaluate an expression tree.

Chapter-3

Flag Word and FLAGS Register (Computing)

Flag word

A **flag word** is a generic term for a word (value) used to indicate conditions within a binary computer. In particular, the byte can be used to show up to 8 discrete conditions. Essentially, each bit represents one flag, capable of showing two states.

A flag word can be seen as a bit array with a short, constant length.

Examples

Processor status register

Taking the example of a 6502 processor's status register, the following information was held within 8 bits:

- Bit 7. Negative flag
- Bit 6. Overflow flag
- Bit 5. Unused
- Bit 4. Break flag
- Bit 3. Decimal flag
- Bit 2. Interrupt-disable flag
- Bit 1. Carry flag
- Bit 0. Zero flag

As you can see, a lot of information about the state of the processor can be packed into 8 bits.

Unix process exit code

A more general example would be the use of a Unix exit status as a flag word. In this case, the exit code is by the programmer to pass status information to another process. An imaginary program which returns the status of 8 burglar alarm switches connected to the printer port could set each of the bits in the exit code in turn, depending on whether the switches are closed or open.

Extracting bits from flag words

To read a status byte, assuming your programming language does not offer this facility by default, is quite easy. You simply need to AND the status byte with a mask byte. The mask byte should have only the bit corresponding to the flag you want to read set, as in the example below.

Suppose that status byte 103 (decimal) is returned, and that we want to check flag bit 5.

The flag we want to read is number 5 (counting from zero) - so the mask byte will be 2^5 = 32. ANDing 32 with 103 gives 32, which means the flag bit is set. If the flag bit was not set, the result would have been 0.

In modern computing, the shift operator (<<) can be used to quickly perform the power-of-two. In general, a mask with the Nth bit set can be computed as

```
(1 << n)
```

Thus to check the Nth bit from a variable **v**, we can perform the operation

```
bool nth_is_set = (v & (1 << n)) != 0
```

Changing bits in flag words

Writing, reading or toggling bits in flags can be done only using the OR, AND and NOT operations - operations which can be performed quickly in the processor.

To set a bit, OR the status byte with a mask byte. Any bits set in the mask byte or the status byte will be set in the result.

```
int setBit(int val, int bit_position)
{
  return val | (1 << bit_position);
}
```

To clear a bit, perform a NOT operation on the mask byte, then AND it with the status byte. The result will have the appropriate flag cleared (set to 0).

```
int clearBit(int val, int bit_position)
```

```
{
  return val & ~(1 << bit_position);
}
```

To toggle a bit, XOR the status byte and the mask byte. This will set a bit if it is cleared or clear a bit if it is set.

```
int toggleBit(int val, int bit_position)
{
  return val ^ (1 << bit_position);
}
```

FLAGS register (computing)

The **FLAGS** register is the status register in Intel x86 microprocessors that contains the current state of the processor. This register is 16-bits wide. Its successors, the **EFLAGS** and **RFLAGS** registers are 32-bits and 64-bits wide, respectively. The wider registers retain compatibility with their smaller predecessors.

Intel x86 FLAGS Register			
Bit #	**Abbreviation**	**Description**	**Category**
FLAGS			
0	CF	Carry flag	S
1	1	Reserved	
2	PF	Parity flag	S
3	0	Reserved	
4	AF	Adjust flag	S
5	0	Reserved	
6	ZF	Zero flag	S
7	SF	Sign flag	S
8	TF	Trap flag (single step)	X
9	IF	Interrupt enable flag	X
10	DF	Direction flag	C
11	OF	Overflow flag	S
12, 13	IOPL	I/O privilege level (286+ only)	X
14	NT	Nested task flag (286+ only)	X
15	0	Reserved	
EFLAGS			
16	RF	Resume flag (386+ only)	X
17	VM	Virtual 8086 mode flag (386+ only)	X
18	AC	Alignment check (486SX+ only)	X

19	VIF	Virtual interrupt flag (Pentium+)	X
20	VIP	Virtual interrupt pending (Pentium+)	X
21	ID	Able to use CPUID instruction (Pentium+)	X
22	0	Reserved	
23	0	Reserved	
24	0	Reserved	
25	0	Reserved	
26	0	Reserved	
27	0	Reserved	
28	0	Reserved	
29	0	Reserved	
30	0	Reserved	
31	0	Reserved	
RFLAGS			
32-63	0	Reserved	

Examples

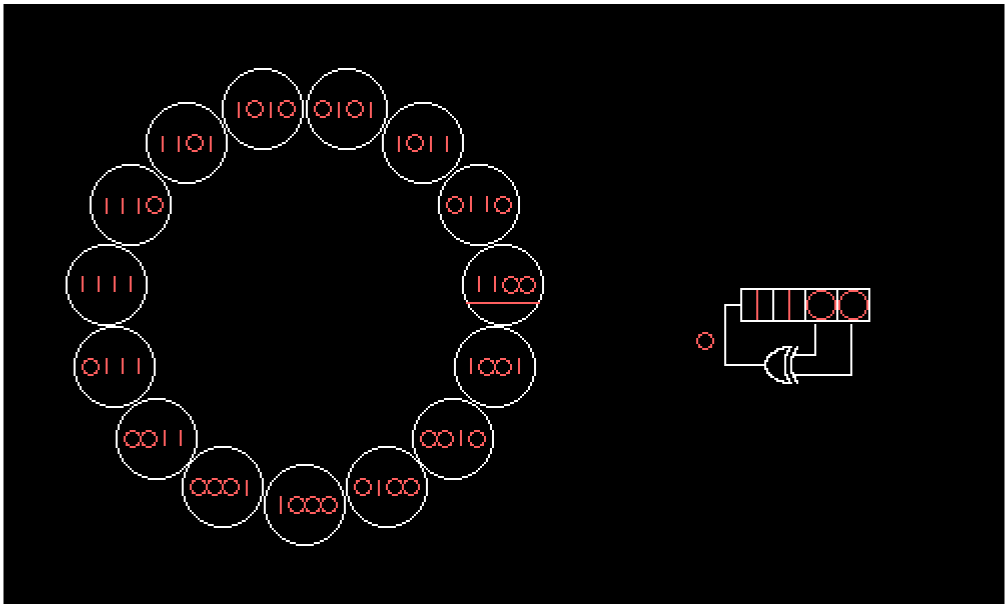
Below is an example for changing the flag in DF (direction flag)

```
mov bx, 400h ; Set the DF flag
pushf ; Pushes the current flags onto the stack
pop ax ; Pop the flags from the stack into ax register
push ax ; Push them back onto the stack for storage
xor ax, bx ; XOR dest, src | Used for toggling the DF flag only, keep
the rest of the flags
push ax ; Push again to add the new value to the stack
popf ; Pop the newly pushed into the FLAGS register
; ... Code here ...
popf ; Pop the old FLAGS back into place

;In practical software, the cld and std mnemonics are used to clear and
set the direction flag, respectively.
```

Chapter-4

Linear Feedback Shift Register



A 4-bit Fibonacci LFSR with its state diagram. The XOR gate provides feedback to the register that shifts bits from left to right. The maximal sequence consists of every possible state except the "0000" state.

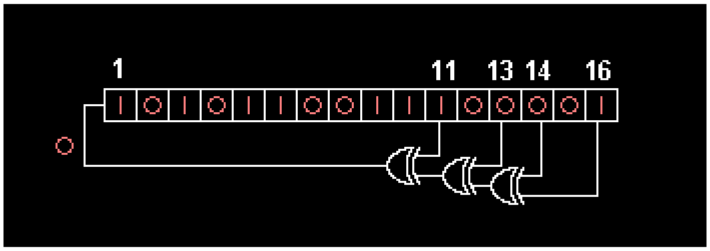
A **linear feedback shift register** (LFSR) is a shift register whose input bit is a linear function of its previous state.

The only linear function of single bits is xor, thus it is a shift register whose input bit is driven by the exclusive-or (xor) of some bits of the overall shift register value.

The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits which appears random and which has a very long cycle.

Applications of LFSRs include generating pseudo-random numbers, pseudo-noise sequences, fast digital counters, and whitening sequences. Both hardware and software implementations of LFSRs are common.

Fibonacci LFSRs



A 16-bit Fibonacci LFSR. The feedback tap numbers in white correspond to a primitive polynomial in the table so the register cycles through the maximum number of 65535 states excluding the all-zeroes state. The state ACE1 hex shown will be followed by 5670 hex.

The bit positions that affect the next state are called the taps. In the diagram the taps are [16,14,13,11]. The rightmost bit of the LFSR is called the output bit. The taps are XOR'd sequentially with the output bit and then fed back into the leftmost bit. The sequence of bits in the rightmost position is called the output stream.

- The bits in the LFSR state which influence the input are called *taps* (white in the diagram).
- A maximum-length LFSR produces an m-sequence (i.e. it cycles through all possible $2^n - 1$ states within the shift register except the state where all bits are zero), unless it contains all zeros, in which case it will never change.
- As an alternative to the XOR based feedback in an LFSR, one can also use XNOR. This function is not linear, but it results in an equivalent polynomial counter whose state of this counter is the complement of the state of an LFSR. A state with all ones is illegal when using an XNOR feedback, in the same way as a state with all zeroes is illegal when using XOR. This state is considered illegal because the counter would remain "locked-up" in this state.

The sequence of numbers generated by an LFSR or its XNOR counterpart can be considered a binary numeral system just as valid as Gray code or the natural binary code.

The arrangement of taps for feedback in an LFSR can be expressed in finite field arithmetic as a polynomial mod 2. This means that the coefficients of the polynomial must be 1's or 0's. This is called the feedback polynomial or characteristic polynomial.

For example, if the taps are at the 16th, 14th, 13th and 11th bits (as shown), the feedback polynomial is

$$x^{16} + x^{14} + x^{13} + x^{11} + 1.$$

The 'one' in the polynomial does not correspond to a tap — it corresponds to the input to the first bit (i.e. x^0, which is equivalent to 1). The powers of the terms represent the tapped bits, counting from the left. The first and last bits are always connected as an input and tap respectively.

Tables of primitive polynomials from which maximum-length LFSRs can be constructed are given below and in the references.

- The LFSR will only be maximum-length if the number of taps is even; just 2 or 4 taps can suffice even for extremely long sequences.
- The set of taps must be relatively prime, and share no common divisor to all taps.
- There can be more than one maximum-length tap sequence for a given LFSR length
- Once one maximum-length tap sequence has been found, another automatically follows. If the tap sequence, in an n-bit LFSR, is $[n, A, B, C, 0]$, where the 0 corresponds to the $x^0 = 1$ term, then the corresponding 'mirror' sequence is $[n, n - C, n - B, n - A, 0]$. So the tap sequence [32, 7, 3, 2, 0] has as its counterpart [32, 30, 29, 25, 0]. Both give a maximum-length sequence.

Some example C/C++ code is below (assuming 16-bit `short`s):

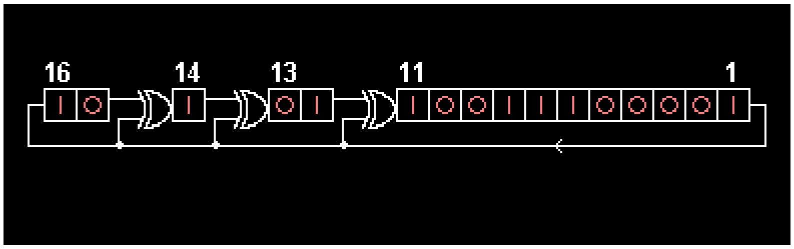
```
unsigned short lfsr = 0xACE1u;
unsigned bit;
unsigned period = 0;
do
{
  /* taps: 16 14 13 11; characteristic polynomial: x^16 + x^14 + x^13 +
x^11 + 1 */
  bit  = (((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1)
^ 1;
  lfsr =  (lfsr >> 1) | (bit << 15);
  ++period;
} while(lfsr != 0xACE1u);
```

The above code assumes the most significant bit of `lfsr` is bit 1, and the least significant bit is bit 16.

As well as *Fibonacci*, this LFSR configuration is also known as **standard**, **many-to-one** or **external XOR gates**. LFSR has an alternative configuration.

Galois LFSRs

Named after the French mathematician Évariste Galois, an LFSR in Galois configuration, which is also known as **modular**, **internal XORs** as well as **one-to-many LFSR**, is an alternate structure that can generate the same output stream as a conventional LFSR. In the Galois configuration, when the system is clocked, bits that are not taps are shifted one position to the right unchanged. The taps, on the other hand, are XOR'd with the output bit before they are stored in the next position. The new output bit is the next input bit. The effect of this is that when the output bit is zero all the bits in the register shift to the right unchanged, and the input bit becomes zero. When the output bit is one, the bits in the tap positions all flip (if they are 0, they become 1, and if they are 1, they become 0), and then the entire register is shifted to the right and the input bit becomes 1.



A 16-bit Galois LFSR. The register numbers in white correspond to the same primitive polynomial as the Fibonacci example but are counted in reverse to the shifting direction. This register also cycles through the maximal number of 65535 states excluding the all-zeroes state. The state ACE1 hex shown will be followed by E270 hex.

To generate the same output stream, the order of the taps is the *counterpart* of the order for the conventional LFSR, otherwise the stream will be in reverse. Note that the internal state of the LFSR is not necessarily the same. The Galois register shown has the same output stream as the Fibonacci register in the first section.

- Galois LFSRs do not concatenate every tap to produce the new input (the XOR'ing is done within the LFSR and no XOR gates are run in serial, therefore the propagation times are reduced to that of one XOR rather than a whole chain), thus it is possible for each tap to be computed in parallel, increasing the speed of execution.
- In a software implementation of an LFSR, the Galois form is more efficient as the XOR operations can be implemented a word at a time: only the output bit must be examined individually.

Below is a code example of a 32-bit maximal period Galois LFSR that is valid in C and C++, (assuming that `unsigned int` has 32 bit precision):

```
unsigned lfsr = 1;
unsigned period = 0;

do {
  /* taps: 32 31 29 1; characteristic polynomial: x^32 + x^31 + x^29 +
x + 1 */
  lfsr = (lfsr >> 1) ^ (unsigned int)(0 - (lfsr & 1u) & 0xd0000001u);
  ++period;
} while(lfsr != 1u);
```

And here is the code for the 16 bit example in the figure (Assuming 16-bit `short`s)

```
unsigned short lfsr = 0xACE1u;
unsigned period = 0;

do {
  /* taps: 16 14 13 11; characteristic polynomial: x^16 + x^14 + x^13 +
x^11 + 1 */
  lfsr = (lfsr >> 1) ^ (-(lfsr & 1u) & 0xB400u);
  ++period;
} while(lfsr != 0xACE1u);
```

These code examples create a toggle mask to apply to the shifted value using the XOR operator. The mask is created by first removing all but the least significant bit (the output bit) of the current value. This value is then negated (two's complement negation), which creates a value of either all 0s or all 1s, if the output bit is 0 or 1, respectively. By ANDing the result with the tap-value (e.g., 0xB400 in the second example) before applying it as the toggle mask, it acts functionally as a conditional to either apply or not apply the toggle mask based on the output bit. A more explicit but significantly less efficient code example is shown below.

```
unsigned short lfsr = 0xACE1u;
unsigned period = 0;

do {
  unsigned lsb = lfsr & 1;  /* Get lsb (i.e., the output bit). */
  lfsr >>= 1;               /* Shift register */
  if (lsb == 1)             /* Only apply toggle mask if output bit is
1. */
    lfsr ^= 0xB400u;        /* Apply toggle mask, value has 1 at bits
corresponding
                             * to taps, 0 elsewhere. */
  ++period;
} while(lfsr != 0xACE1u);
```

Non-binary Galois LFSR

Binary Galois LFSRs like the ones shown above can be generalized to any q-ary alphabet $\{0, 1, \ldots, q - 1\}$ (e.g., for binary, q is equal to two, and the alphabet is simply $\{0, 1\}$). In this case, the exclusive-or component is generalized to addition modulo-q (note that XOR is addition modulo 2), and the feedback bit (output bit) is multiplied (modulo-q) by a q-ary value which is constant for each specific tap point. Note that this is also a generalization of the binary case, where the feedback is multiplied by either 0 (no feedback, i.e., no tap) or 1 (feedback is present). Given an appropriate tap configuration, such LFSRs can be used to generate Galois fields for arbitrary values of q.

Some polynomials for maximal LFSRs

The following table lists maximal-length polynomials for shift-register lengths up to 19. Note that more than one maximal-length polynomial may exist for any given shift-register length.

Bits	**Feedback polynomial**	**Period**
n		$2^n - 1$
2	$x^2 + x + 1$	3
3	$x^3 + x^2 + 1$	7
4	$x^4 + x^3 + 1$	15
5	$x^5 + x^3 + 1$	31
6	$x^6 + x^5 + 1$	63
7	$x^7 + x^6 + 1$	127
8	$x^8 + x^6 + x^5 + x^4 + 1$	255
9	$x^9 + x^5 + 1$	511
10	$x^{10} + x^7 + 1$	1023
11	$x^{11} + x^9 + 1$	2047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	4095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	8191
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	16383
15	$x^{15} + x^{14} + 1$	32767
16	$x^{16} + x^{14} + x^{13} + x^{11} + 1$	65535
17	$x^{17} + x^{14} + 1$	131071
18	$x^{18} + x^{11} + 1$	262143
19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	524287
20 to 168		

Output-stream properties

- Ones and zeroes occur in 'runs'. The output stream 0110100, for example consists of five runs of lengths 1,2,1,1,2, in order. In one period of a maximal LFSR, 2^{n-1} runs occur (for example, a six bit LFSR will have 32 runs). Exactly 1/2 of these runs will be one bit long, 1/4 will be two bits long, up to a single run of zeroes $n - 1$ bits long, and a single run of ones n bits long. This distribution almost equals the statistical expectation value for a truly random sequence. However, the probability of finding exactly this distribution in a sample of a truly random sequence is rather low.
- LFSR output streams are deterministic. If you know the present state, you can predict the next state. This is not possible with truly random events.
- The output stream is reversible; an LFSR with mirrored taps will cycle through the output sequence in reverse order.

Applications

LFSRs can be implemented in hardware, and this makes them useful in applications that require very fast generation of a pseudo-random sequence, such as direct-sequence spread spectrum radio. LFSRs have also been used for generating an approximation of white noise in various programmable sound generators.

The Global Positioning System uses an LFSR to rapidly transmit a sequence that indicates high-precision relative time offsets.

Uses as counters

The repeating sequence of states of an LFSR allows it to be used as a clock divider, or as a counter when a non-binary sequence is acceptable as is often the case where computer index or framing locations need to be machine-readable. LFSR counters have simpler feedback logic than natural binary counters or Gray code counters, and therefore can operate at higher clock rates. However it is necessary to ensure that the LFSR never enters an all-zeros state, for example by presetting it at start-up to any other state in the sequence. The table of primitive polynomials shows how LFSRs can be arranged in Fibonacci or Galois form to give maximal periods. One can obtain any other period by adding to an LFSR that has a longer period some logic that shortens the sequence by skipping some states.

Uses in cryptography

LFSRs have long been used as pseudo-random number generators for use in stream ciphers (especially in military cryptography), due to the ease of construction from simple electromechanical or electronic circuits, long periods, and very uniformly distributed output streams. However, an LFSR is a linear system, leading to fairly easy cryptanalysis. For example, given a stretch of known plaintext and corresponding ciphertext, an attacker can intercept and recover a stretch of LFSR output stream used in the system described, and from that stretch of the output stream can construct an LFSR of minimal size that simulates the intended receiver by using the Berlekamp-Massey algorithm. This LFSR can then be fed the intercepted stretch of output stream to recover the remaining plaintext.

Three general methods are employed to reduce this problem in LFSR-based stream ciphers:

- Non-linear combination of several bits from the LFSR state;
- Non-linear combination of the output bits of two or more LFSRs
- Irregular clocking of the LFSR, as in the alternating step generator.

Important LFSR-based stream ciphers include A5/1 and A5/2, used in GSM cell phones, E0, used in Bluetooth, and the shrinking generator. The A5/2 cipher has been broken and both A5/1 and E0 have serious weaknesses.

Uses in digital broadcasting and communications

To prevent short repeating sequences (e.g., runs of 0's or 1's) from forming spectral lines that may complicate symbol tracking at the receiver or interfere with other transmissions, linear feedback registers are often used to "randomize" the transmitted bitstream. This randomization is removed at the receiver after demodulation. When the LFSR runs at the same rate as the transmitted symbol stream, this technique is referred to as scrambling. When the LFSR runs considerably faster than the symbol stream, expanding the bandwidth of the transmitted signal, this is direct-sequence spread spectrum.

Neither scheme should be confused with encryption or encipherment; scrambling and spreading with LFSRs do *not* protect the information from eavesdropping. They are instead used to produce equivalent streams that possess convenient engineering properties to allow for robust and efficient modulation and demodulation.

Digital broadcasting systems that use linear feedback registers:

- ATSC Standards (digital TV transmission system – North America)
- DAB (Digital Audio Broadcasting system – for radio)
- DVB-T (digital TV transmission system – Europe, Australia, parts of Asia)
- NICAM (digital audio system for television)

Other digital communications systems using LFSRs:

- IBS (INTELSAT business service)
- IDR (Intermediate Data Rate service)
- SDI (Serial Digital Interface transmission)
- Data transfer over PSTN (according to the ITU-T V-series recommendations)
- CDMA (Code Division Multiple Access) cellular telephony
- 100BASE-T2 "fast" Ethernet scrambles bits using an LFSR
- 1000BASE-T Ethernet, the most common form of Gigabit Ethernet, scrambles bits using an LFSR
- PCI Express 3.0
- USB 3.0
- IEEE 802.11a scrambles bits using an LFSR

The mathematics of a cyclic redundancy check, used to provide a quick check against transmission errors, are closely related to those of an LFSR.

Chapter-5

Program Counter and Register File

Program counter

The **program counter**, or PC (also called the **instruction pointer** to a seminal Intel instruction set, such as the 8080 or 4004, or **instruction address register**, or just part of the **instruction sequencer** in some computers) is a processor register that indicates where the computer is in its instruction sequence. Depending on the details of the particular computer, the PC holds either the **address of the instruction being executed**, or **the address of the next instruction to be executed**.

In most processors, the program counter is incremented automatically after fetching a program instruction, so that instructions are normally retrieved sequentially from memory, with certain instructions, such as branches, jumps and subroutine calls and returns, interrupting the sequence by placing a new value in the program counter.

Such jump instructions allow a new address to be chosen as the start of the next part of the flow of instructions from the memory. They allow new values to be loaded (written) into the program counter register. A subroutine call is achieved simply by reading the old contents of the program counter, before they are overwritten by a new value, and saving them somewhere in memory or in another register. A subroutine return is then achieved by writing the saved value back in to the program counter again.

Working of a simple program counter

The central processing unit (CPU) of a simple computer contains the hardware (control unit and ALU) that executes the instructions, as they are fetched from the memory unit. Most instruction cycles consist of the CPU sending an **address**, on the address bus, to the **memory unit**, which then responds by sending the contents of that memory location as **data**, on the data bus. (This is tied up with the idea of the stored-program computer in which executable instructions are stored alongside ordinary data in the **memory unit**, and handled identically by it). The Program Counter (PC) is just one of the many registers in the hardware of the CPU. It, like each of the other registers, consists of a bank of binary latches (a binary latch is also known as a flip-flop), with one flip-flop per bit in the integer that is to be stored (32 for a 32-bit CPU, for example). In the case of the PC, the integer represents the address in the **memory unit** that is to be fetched next.

Once the data (the instruction) has been received on the **data bus**, the PC is incremented. In some CPUs this is achieved by adding 000..001 to its contents, and latching the result into the register to be its new contents; on most CPUs, though, the PC is implemented as a register that is internally wired so that it counts up to the next value when a specific signal is applied to it externally. Such a register, in electronics, is referred to as a binary counter, and hence the origin of the term **program counter**.

The all-pervading nature of the program counter

The presence of the program counter in the CPU has far reaching consequences on our way of thinking when we program computers. Indeed, the program counter (or any equivalent block of hardware that serves the same purpose) is very much central to the von Neumann architecture.

The PC imposes a strict sequential ordering on the fetching of instructions from the memory unit (the flow of control), even where no sequentiality is implied by the algorithm itself (the von Neumann bottleneck). This is why research into possible models for parallel computing considered, at one point, other **non von Neumann** or dataflow models that did not use a program counter. For example, functional programming languages offered much hope at the high level, with combinatory logic at the assembler level. Even then, most of the researchers emulated this in the microcode of conventional computers (hence still involving a program counter in the hardware); but, in fact, combinators are so simple, they could, in principle, be implemented directly in the hardware without recourse to microcode or program counters at all.

In the end, though, the results of that research fed back, instead, into ways of improving the execution speed of conventional processors. Ways were found for organising out-of-order execution so as to extract the sequencing information that is implicit in the data. Also, the pipeline and very long instruction word organisations allowed the compiler to arrange for multiple calculations to be set off in parallel. At the start of each instruction execution, though, the instruction needs to be fetched from memory, and this is initiated by an **instruction fetch** cycle that gets the instructions, one at a time, as directed by the program counter.

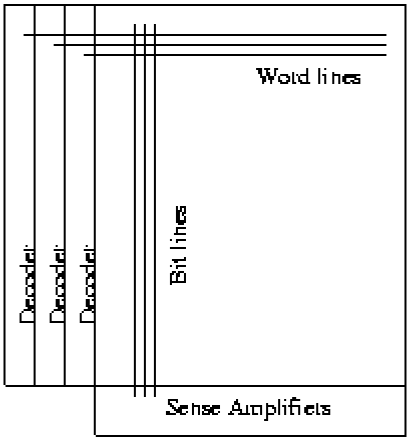
Even high level programming languages have the program-counter concept engrained deep down in their behavior. You need only to watch how a programmer develops or debugs a computer program to see evidence of this, with the programmer using a finger to point to successive lines in the program to model the steps of its execution. Indeed, a high level programming language is no less than the assembler language of a high level virtual machine -- a computer that would be too complex to be cost-effective to build directly in hardware, so is implemented instead using multiple shells of emulation (with the compiler or interpreter providing the higher levels, and the microcode providing the lower levels).

Register file

A **register file** is an array of processor registers in a central processing unit (CPU). Modern integrated circuit-based register files are usually implemented by way of fast static RAMs with multiple ports. Such RAMs are distinguished by having dedicated read and write ports, whereas ordinary multiported SRAMs will usually read and write through the same ports.

The instruction set architecture of a CPU will almost always define a set of registers which are used to stage data between memory and the functional units on the chip. In simpler CPUs, these *architectural registers* correspond one-for-one to the entries in a physical register file within the CPU. More complicated CPUs use register renaming, so that the mapping of which physical entry stores a particular architectural register changes dynamically during execution. The register file is part of the architecture and visible to the programmer, as opposed to the concept of transparent caches.

Implementation



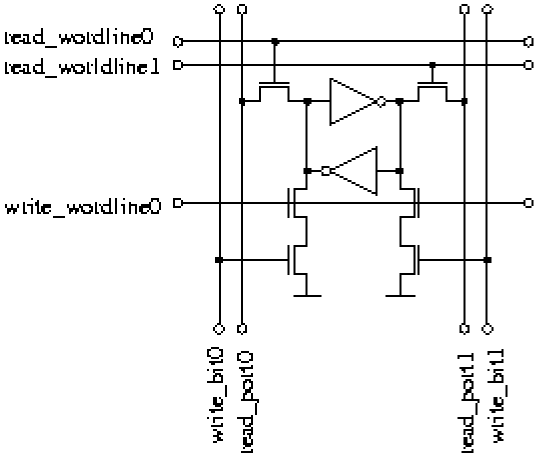
The usual layout convention is that a simple array is read out vertically. That is, a single word line, which runs horizontally, causes a row of bit cells to put their data on bit lines, which run vertically. Sense amps, which convert low-swing read bitlines into full-swing logic levels, are usually at the bottom (by convention). Larger register files are then sometimes constructed by tiling mirrored and rotated simple arrays.

Register files have one word line per entry per port, one bit line per bit of width per read port, and two bit lines per bit of width per write port. Each bit cell also has a Vdd and Vss. Therefore, the wire pitch area increases as the square of the number of ports, and the transistor area increases linearly. At some point, it may be smaller and/or faster to have multiple redundant register files, with smaller numbers of read ports, than a single register file with all the read ports. The MIPS R8000's integer unit, for example, had a 9 read 4 write port 32 entry 64-bit register file implemented in a 0.7 µm process, which could be seen when looking at the chip from arm's length.

Decoder

- The decoder is often broken into predecoder and decoder proper.
- The decoder is a series of AND gates that drive word lines.
- There is one decoder per read or write port. If the array has four read and two write ports, for example, it has 6 word lines per bit cell in the array, and six AND gates per row in the decoder. Note that the decoder has to be pitch matched to the array, which forces those AND gates to be wide and short

Array



A typical register file -- "triple-ported", able to read from 2 registers and write to 1 register simultaneously -- is made of bit cells like this one.

The basic scheme for a bit cell:

- State is stored in pair of inverters
- Data is read out by nmos transistor to a bit line.
- Data is written by shorting one side or the other to ground through a two-nmos stack.
- So: read ports take one transistor per bit cell, write ports take four!

Many optimizations are possible:

- Sharing lines between cells, for example, Vdd and Vss.
- Read bit lines are often precharged to something between Vdd and Vss.
- Read bit lines often swing only a fraction of the way to Vdd or Vss. A sense amplifier converts this small-swing signal into a full logic level. Small swing signals are faster because the bit line has little drive but a great deal of parasitic capacitance.

- Write bit lines may be braided, so that they couple equally to the nearby read bitlines. Because write bitlines are full swing, they can cause significant disturbances on read bitlines.
- If Vdd is a horizontal line, it can be switched off, by yet another decoder, if any of the write ports are writing that line during that cycle. This optimization increases the speed of the write.
- Techniques that reduce the energy used by register files are useful in low-power electronics

Microarchitecture

Most register files make no special provision to prevent multiple write ports from writing the same entry simultaneously. Instead, the instruction scheduling hardware ensures that only one instruction in any particular cycle writes a particular entry. If multiple instructions targeting the same register are issued, all but one have their write enables turned off.

The crossed inverters take some finite time to settle after a write operation, during which a read operation will either take longer or return garbage. It is common to have bypass multiplexors that bypass written data to the read ports when a simultaneous read and write to the same entry is commanded. These bypass multiplexors are often just part of a larger bypass network that forwards results that have not yet been committed between functional units.

The register file is usually pitch matched to the datapath that it serves. Pitch matching avoids having the many busses passing over the datapath turn corners, which would use a lot of area. But since every unit must have the same bit pitch, every unit in the datapath ends up with the bit pitch forced by the widest unit, which can waste area in the other units. Register files, because they have two wires per bit per write port, and because all the bit lines must contact the silicon at every bit cell, can often set the pitch of a datapath.

Area can sometimes be saved, on machines with multiple units in a datapath, by having two datapaths side-by-side, each of which has smaller bit pitch than a single datapath would have. This case usually forces multiple copies of a register file, one for each datapath.

The Alpha 21264 (EV6), for instance, had two copies of the integer register file, and took an extra cycle to propagate data between the two. The issue logic tried to reduce the number of operations forwarding data between the two. The MIPS R8000 floating-point unit had two copies of the floating-point register file, each with four write and four read ports, and wrote both copies at the same time.

Processors that do register renaming can arrange for each functional unit to write to a subset of the physical register file. This arrangement can eliminate the need for multiple write ports per bit cell, for large savings in area. The resulting register file, effectively a stack of register files with single write ports, then benefits from replication and subsetting the read ports. At the limit, this technique would place a stack of 1-write, 2-read regfiles at the inputs to each functional unit. Since regfiles with a small number of ports are often dominated by transistor area, it is best not to push this technique to this limit, but it is useful all the same.

The SPARC ISA defines register windows, in which the 5-bit architectural names of the registers actually point into a window on a much larger register file, with hundreds of entries. Implementing multiported register files with hundreds of entries requires a lot of area. The register window slides by 16 registers when moved, so that each architectural register name can refer to only a small number of registers in the larger array, e.g. architectural register r20 can only refer to physical registers #20, #36, #52, #68, #84, #100, #116, if there are just seven windows in the physical file.

To save area, some SPARC implementations implement a 32-entry register file, in which each cell has seven "bits". Only one is read and writeable through the external ports, but the contents of the bits can be rotated. A rotation accomplishes in a single cycle a movement of the register window. Because most of the wires accomplishing the state movement are local, tremendous bandwidth is possible with little power.

This same technique is used in the R10000 register renaming mapping file, which stores a 6-bit virtual register number for each of the physical registers. In the renaming file, the renaming state is checkpointed whenever a branch is taken, so that when a branch is detected to be mispredicted, the old renaming state can be recovered in a single cycle.

Chapter-6

Register Renaming

In computer architecture, **register renaming** refers to a technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations.

Problem definition

Programs are composed of instructions which operate on values. The instructions must name these values in order to distinguish them from one another. A typical instruction might say, add X and Y and put the result in Z. In this instruction, X, Y, and Z are the names of storage locations.

In order to have a compact instruction encoding, most processor instruction sets have a small set of special locations which can be directly named. For example, the x86 instruction set architecture has 8 integer registers, x86-64 has 16, many RISCs have 32, and IA-64 has 128. In smaller processors, the names of these locations correspond directly to elements of a register file.

Different instructions may take different amounts of time (e.g., CISC architecture). For instance, a processor may be able to execute hundreds of instructions while a single load from main memory is in progress. Shorter instructions executed while the load is outstanding will finish first, thus the instructions are finishing out of the original program order. Out-of-order execution has been used in most recent high-performance CPUs to achieve some of their speed gains.

Consider this piece of code running on an out-of-order CPU:

1. R1=M[1024]
2. R1=R1+2

3. M[1032]=R1
4. R1=M[2048]
5. R1=R1+4
6. M[2056]=R1

Instructions 4, 5, and 6 are independent of instructions 1, 2, and 3, but the processor cannot finish 4 until 3 is done, because 3 would then write the wrong value.

We can eliminate this restriction by changing the names of some of the registers:

1. R1=M[1024]	4. R2=M[2048]
2. R1=R1+2	5. R2=R2+4
3. M[1032]=R1	6. M[2056]=R2

Now instructions 4, 5, and 6 can be executed in parallel with instructions 1, 2, and 3, so that the program can be executed faster.

When possible, the compiler performs this renaming. The compiler is constrained in many ways, primarily by the finite number of register names in the instruction set. Many high performance CPUs have more physical registers than may be named directly in the instruction set, so they rename registers in hardware to achieve additional parallelism.

Data hazards

When more than one instruction references a particular location for an operand, either reading it (as an input) or writing it (as an output), executing those instructions in an order different from the original program order can lead to three kinds of data hazards:

Read-after-write (RAW)

A read from a register or memory location must return the value placed there by the last write in program order, not some other write. This is referred to as a **true dependency** or **flow dependency**, and requires the instructions to execute in program order.

Write-after-write (WAW)

Successive writes to a particular register or memory location must leave that location containing the result of the second write. This can be resolved by **squashing** (synonyms: cancelling, annulling, mooting) the first write if necessary. WAW dependencies are also known as **output dependencies**.

Write-after-read (WAR)

A read from a register or memory location must return the last prior value written to that location, and not one written programmatically after the read. This is the sort of **false**

dependency that can be resolved by renaming. WAR dependencies are also known as **anti-dependencies**.

Instead of delaying the write until all reads are completed, two copies of the location can be maintained, the old value and the new value. Reads that precede, in program order, the write of the new value can be provided with the old value, even while other reads that follow the write are provided with the new value. The false dependency is broken and additional opportunities for out-of-order execution are created. When all reads needing the old value have been satisfied, it can be discarded. This is the essential concept behind register renaming.

Anything that is read and written can be renamed. While the general-purpose and floating-point registers are discussed the most, flag and status registers or even individual status bits are commonly renamed as well.

Memory locations can also be renamed, although it is not commonly done to the extent practised in register renaming. The Transmeta Crusoe processor's gated store buffer is a form of memory renaming.

If programs refrained from reusing registers immediately, there would be no need for register renaming. Some instruction sets (e.g., IA-64) specify very large numbers of registers for specifically this reason. There are limitations to this approach:

- It is very difficult for the compiler to avoid reusing registers without large code size increases. In loops, for instance, successive iterations would have to use different registers, which requires replicating the code in a process called loop unrolling
- Large numbers of registers require lots of bits to specify those registers, making the code size increase.
- Many instruction sets historically specified smaller numbers of registers and cannot be changed now.

Code size increases are important because when the program code is larger, the instruction cache misses more often and the processor stalls waiting for new instructions.

Architectural vs physical registers

Machine language programs specify reads and writes to a limited set of registers specified by the instruction set architecture (ISA). For instance, the Alpha ISA specifies 32 integer registers, each 64 bits wide, and 32 floating-point registers, each 64 bits wide. These are the **architectural** registers. Programs written for processors running the Alpha instruction set will specify operations reading and writing those 64 registers. If a programmer stops the program in a debugger, she or he can observe the contents of these 64 registers (and a few status registers) to determine the progress of the machine.

One particular processor which implements this ISA, the Alpha 21264, has 80 integer and 72 floating-point **physical** registers. There are, on an Alpha 21264 chip, 80 physically separate locations which can store the results of integer operations, and 72 locations which can store the results of floating point operations. (In fact, there are even more locations than that, but those extra locations are not germane to the register renaming operation.)

Below are described two styles of register renaming, distinguished by the circuit which holds data ready for an execution unit.

In all renaming schemes, the machine converts the architectural registers referenced in the instruction stream into tags. Where the architectural registers might be specified by 3 to 5 bits, the tags are usually a 6 to 8 bit number. The rename file must have a read port for every input of every instruction renamed every cycle, and a write port for every output of every instruction renamed every cycle. Because the size of a register file generally grows as the square of the number of ports, the rename file is usually physically large and consumes significant power.

In the **tag-indexed register file** style, there is one large register file for data values, containing one register for every tag. For example, if the machine has 80 physical registers, then it would use 7 bit tags. 48 of the possible tag values in this case are unused.

In this style, when an instruction is issued to an execution unit, the tags of the source registers are sent to the physical register file, where the values corresponding to those tags are read and sent to the execution unit.

In the **reservation station** style, there are many small associative register files, usually one at the inputs to each execution unit. Each operand of each instruction in an issue queue has a place for a value in one of these register files.

In this style, when an instruction is issued to an execution unit, the register file entries corresponding to the issue queue entry are read and forwarded to the execution unit.

Architectural Register File or Retirement Register File (RRF)

The committed register state of the machine. RAM indexed by logical register number. Typically written into as results are retired or committed out of a reorder buffer.

Future File

The most speculative register state of the machine. RAM indexed by logical register number.

Active Register File

The Intel P6 group's term for Future File.

History Buffer

Typically used in combination with a future file. Contains the "old" values of registers that have been overwritten. If the producer is still in flight it may be

RAM indexed by history buffer number. After a branch misprediction must use results from the history buffer—either they are copied, or the future file lookup is disabled and the history buffer is CAM indexed by logical register number.

Reorder Buffer (ROB)

A structure that is sequentially (circularly) indexed on a per-operation basis, for instructions in flight. It differs from a history buffer because the reorder buffer typically comes after the future file (if it exists) and before the architectural register file.

Reorder buffers come in data-less and data-ful versions.

In Willamette's ROB, the ROB entries point to registers in the physical register file (PRF), and also contain other book keeping. This was also the first Out of Order design done by Andy Glew, at Illinois with HaRRM.

P6's ROB, the ROB entries contain data; there is no separate PRF. Data values from the ROB are copied from the ROB to the RRF at retirement.

One small detail: if there is temporal locality in ROB entries (i.e., if instructions close together in the Von Neuman instruction sequence write back close together in time, it may be possible to perform write combining on ROB entries and so have fewer ports than a separate ROB/PRF would). It is not clear if it makes a difference, since a PRF should be banked.

ROBs usually don't have associative logic, and certainly none of the ROBs designed by Andy Glew have CAMs. Keith Diefendorff insisted that ROBs have complex associative logic for many years. The first ROB proposal may have had CAMs.

Details: tag-indexed register file

This is the renaming style used in the MIPS R10000, the Alpha 21264, and in the FP section of the AMD Athlon.

In the renaming stage, every architectural register referenced (for read or write) is looked up in an architecturally-indexed **remap file**. This file returns a tag and a ready bit. The tag is non-ready if there is a queued instruction which will write to it that has not yet executed. For read operands, this tag takes the place of the architectural register in the instruction. For every register write, a new tag is pulled from a **free tag FIFO**, and a new mapping is written into the remap file, so that future instructions reading the architectural register will refer to this new tag. The tag is marked as unready, because the instruction has not yet executed. The previous physical register allocated for that architectural register is saved with the instruction in the **reorder buffer**, which is a FIFO that holds the instructions in program order between the decode and graduation stages.

The instructions are then placed in various **issue queues**.

As instructions are executed, the tags for their results are broadcast, and the issue queues match these tags against the tags of their non-ready source operands. A match means that the operand is ready. The remap file also matches these tags, so that it can mark the corresponding physical registers as ready.

When all the operands to an instruction in an issue queue are ready, that instruction is ready to issue. The issue queues pick ready instructions to send to the various functional units each cycle. Non-ready instructions stay in the issue queues. This unordered removal of instructions from the issue queues is one of the things that makes them large and use lots of power.

Issued instructions read from a tag-indexed physical register file (bypassing just-broadcast operands), then execute.

Execution results are written to tag-indexed physical register file, as well as broadcast to the bypass network preceding each functional unit.

Graduation puts the previous tag for the written architectural register into the free queue so that it can be reused for a newly decoded instruction.

An exception or branch misprediction causes the remap file to back up to the remap state at last valid instruction via combination of state snapshots and cycling through the previous tags in the in-order pre-graduation queue. Since this mechanism is required, and since it can recover any remap state (not just the state before the instruction currently being graduated), branch mispredictions can be handled before the branch reaches graduation, potentially hiding the branch misprediction latency.

Details: reservation stations

This is the style used in the integer section of the AMD K7 and K8 designs.

In the renaming stage, every architectural register referenced for reads is looked up in both the architecturally-indexed **future file** and the rename file. The future file read gives the value of that register, if there is no outstanding instruction yet to write to it (i.e., it's ready). When the instruction is placed in an issue queue, the values read from the future file are written into the corresponding entries in the reservation stations. Register writes in the instruction cause a new, non-ready tag to be written into the rename file. The tag number is usually serially allocated in instruction order—no free tag FIFO is necessary.

Just as with the tag-indexed scheme, the issue queues wait for non-ready operands to see matching tag broadcasts. Unlike the tag-indexed scheme, matching tags cause the corresponding broadcast value to be written into the issue queue entry's reservation station.

Issued instructions read their arguments from the reservation station, bypass just-broadcast operands, and then execute. As mentioned earlier, the reservation station register files are usually small, with perhaps eight entries.

Execution results are written to the reorder buffer, to the reservation stations (if the issue queue entry has a matching tag), and to the future file if this is the last instruction to target that architectural register (in which case register is marked ready).

Graduation copies the value from the reorder buffer into the architectural register file. The sole use of the architectural register file is to recover from exceptions and branch mispredictions.

Exceptions and branch mispredictions, recognised at graduation, cause the architectural file to be copied to the future file, and all registers marked as ready in the rename file. There is usually no way to reconstruct the state of the future file for some instruction intermediate between decode and graduation, so there is usually no way to do early recovery from branch mispredictions.

Comparison between the schemes

In both schemes, instructions are inserted in-order into the issue queues, but are removed out-of-order. If the queues do not collapse empty slots, then they will either have many unused entries, or require some sort of variable priority encoding for when multiple instructions are simultaneously ready to go. Queues that collapse holes have simpler priority encoding, but require simple but large circuitry to advance instructions through the queue.

Reservation stations have better latency from rename to execute, because the rename stage finds the register values directly, rather than finding the physical register number, and then using that to find the value. This latency shows up as a component of the branch mispredict latency.

Reservation stations also have better latency from instruction issue to execution, because each local register file is smaller than the large central file of the tag-indexed scheme. Tag generation and exception processing are also simpler in the reservation station scheme, as discussed below.

The physical register files used by reservation stations usually collapse unused entries in parallel with the issue queue they serve, which makes these register files larger in aggregate, and burn more power, and more complicated than the simpler register files used in a tag-indexed scheme. Worse yet, every entry in each reservation station can be written by every result bus, so that a reservation-station machine with, e.g., 8 issue queue entries per functional unit will typically have 9 times as many bypass networks as an equivalent tag-indexed machine. Result forwarding thus takes much more power and area than in a tag-indexed design.

Furthermore, the reservation station scheme has four places (Future File, Reservation Station, Reorder Buffer and Architectural File) where a result value can be stored, where the tag-indexed scheme has just one (the physical register file). Because the results from the functional units, broadcast to all these storage locations, must reach a much larger number of locations in the machine than in the tag-indexed scheme, this function consumes more power, area, and time. Still, in machines equipped with very accurate branch prediction schemes and if execute latencies are a major concern, reservation stations can work remarkably well.

History

The IBM System/360 Model 91 was an early machine that supported out-of-order execution of instructions; it used the Tomasulo algorithm, which uses register renaming.

The POWER1 is the first microprocessor that used register renaming and out-of-order execution in 1990.

The original R10000 design had neither collapsing issue queues nor variable priority encoding, and suffered starvation problems as a result—the oldest instruction in the queue would sometimes not be issued until both instruction decode stopped completely for lack of rename registers, and every other instruction had been issued. Later revisions of the design starting with the R12000 used a partially variable priority encoder to mitigate this problem.

Early out-of-order machines did not separate the renaming and ROB/PRF storage functions. For that matter, some of the earliest, such as Sohi's RUU or the Metaflow DCAF, combined scheduling, renaming, and storage all in the same structure.

Most modern machines do renaming by RAM indexing a map table with the logical register number. E.g., P6 did this; future files do this, and have data storage in the same structure.

However, earlier machines used content-addressable memory (a type of hardware which provides the functionality of an associative array) in the renamer. E.g., the HPSM RAT, or Register Alias Table, essentially used a CAM on the logical register number in combination with different versions of the register.

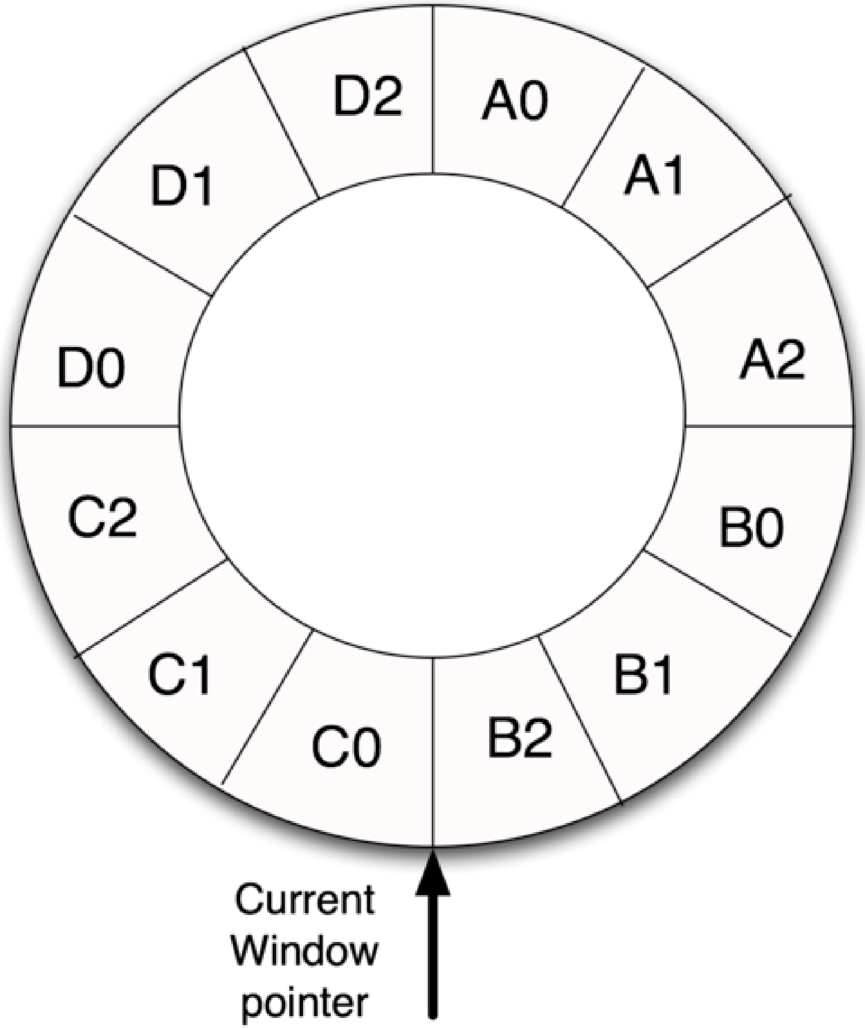
In many ways, the story of out-of-order microarchitecture has been how these CAMs have been progressively eliminated. Small CAMs are useful; large CAMs are impractical.

The P6 microarchitecture was the first Intel based processor that implemented both out-of-order execution and register renaming. The P6 microarchitecture manifested in Pentium Pro, Pentium II, Pentium III, Pentium M, Core, and Core 2 microprocessors.

Chapter-7

Register Window and Shift Register

Register window



Example of a 4-window register window system

In computer engineering, the use of **register windows** is a technique to improve the performance of a particularly common operation, the procedure call. This was one of the main design features of the original Berkeley RISC design, which would later be commercialized as the SPARC, AMD 29000, and Intel i960.

Most CPU designs include a small amount of very high-speed memory known as registers. Registers are used by the CPU in order to hold temporary values while working on longer strings of instructions. Considerable performance can be added to a design with more registers, however, since the registers are a visible piece of the CPU's instruction set, the number cannot typically be changed after the design has been released.

While registers are almost a universal solution to performance, they do have a drawback. Different parts of a computer program all use their own temporary values, and therefore compete for the use of the registers. Since a good understanding of the nature of program flow at runtime is very difficult, there is no easy way for the developer to know in advance how many registers they should use, and how many to leave aside for other parts of the program. In general these sorts of considerations are ignored, and the developers, and more likely, the compilers they use, attempt to use all the registers visible to them. (In the case of processors with very few registers to begin with, this is also the only reasonable course of action.)

This is where register windows become useful. Since every part of a program wants registers for its own use, it makes sense to provide several sets of registers for the different parts of the program. Of course if these registers were visible, there would simply be more registers to compete over, the "trick" is to make them invisible. This is actually somewhat simpler than it might sound; the movement from one part of the program to another during a procedure call is easily "seen", it is accomplished by one of a small number of instructions and ends with one of a similarly small set. In the Berkeley design, these calls would cause a new set of registers to be "swapped in" at that point, or marked as "dead" (or "reusable") when the call ends.

In the Berkeley RISC design, only eight registers were visible to the programs, out of a total of 64. The complete set of registers was known as the register file, and any particular set of eight as a **window**. The file allowed up to eight procedure calls to have their own register sets. As long as the program did not call down chains longer than eight calls deep, the registers never had to be *spilled* (saved out to main memory or cache), a terribly slow process compared to register access. For many programs a chain of six is as deep as the program will go.

By comparison the Sun Microsystems SPARC architecture provides simultaneous visibility into four sets of eight registers each. Three sets of eight registers each are "windowed". Eight registers (i0 through i7) form the input registers to the current procedure level. Eight registers (L0 through L7) are local to the current procedure level, and eight registers (o0 through o7) are the outputs from the current procedure level to the next level called. When a procedure is called, the register window shifts by sixteen registers, hiding the old input registers and old local registers and making the old output registers the new input registers. The common registers (old output registers and new input registers) are used for parameter passing. Finally, eight registers (g0 through g7) are globally visible to all procedure levels.

The AMD 29000 improved the design by allowing the windows to be of variable size, which helps utilization in the common case where fewer than eight registers are needed for a call. It also separated the registers into a global set of 64, and an additional 128 for the windows.

Register windows also provide an easy upgrade path. Since the additional registers are invisible to the programs, additional windows can be added at any time. For instance, the use of object-oriented programming often results in a greater number of "smaller" calls, which can be accommodated by increasing the windows from eight to sixteen for instance. This was the approach used in the SPARC, which has included more register windows with newer generations of the architecture. The end result is fewer slow register window *spill* and *fill* operations because the register windows overflow less often.

Register windows are not the only way to improve register performance. The group at Stanford University designing the MIPS architecture saw the Berkeley work and decided that the problem was not a shortage of registers, but poor utilization of the existing ones. They instead invested more time in their compiler's register allocation, making sure it wisely used the larger set available in the MIPS instruction set. This resulted in reduced complexity of the chip, with one half the total number of registers, while offering potentially higher performance in those cases where a single procedure could make use of the larger visible register space. In the end, with modern compilers, the MIPS design makes better use of its register space even during procedure calls.

Shift register

In digital circuits, a **shift register** is a cascade of flip flops, sharing the same clock, which has the output of any one but the last flip-flop connected to the "data" input of the next one in the chain, resulting in a circuit that shifts by one position the one-dimensional "bit array" stored in it, *shifting in* the data present at its input and *shifting out* the last bit in the array, when enabled to do so by a transition of the clock input. More generally, a **shift register** may be multidimensional, such that its "data in" input and stage outputs are themselves bit arrays: this is implemented simply by running several shift registers of the same bit-length in parallel.

Shift registers can have both parallel and serial inputs and outputs. These are often configured as **serial-in, parallel-out** (SIPO) or as **parallel-in, serial-out** (PISO). There are also types that have both serial and parallel input and types with serial and parallel output. There are also **bi-directional** shift registers which allow shifting in both directions: L→R or R→L. The serial input and last output of a shift register can also be connected together to create a **circular shift register**.

Serial-in, serial-out (SISO)

Destructive readout

0	0	0	0
1	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1
0	1	1	0
0	0	1	1
0	0	0	1
0	0	0	0

These are the simplest kind of shift registers. The data string is presented at 'Data In', and is shifted right one stage each time 'Data Advance' is brought high. At each advance, the bit on the far left (i.e. 'Data In') is shifted into the first flip-flop's output. The bit on the far right (i.e. 'Data Out') is shifted out and lost.

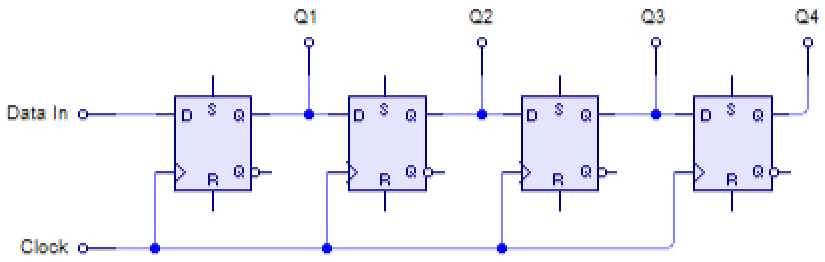
The data are stored after each flip-flop on the 'Q' output, so there are four storage 'slots' available in this arrangement, hence it is a 4-Bit Register. To give an idea of the shifting pattern, imagine that the register holds 0000 (so all storage slots are empty). As 'Data In' presents 1,0,1,1,0,0,0,0 (in that order, with a pulse at 'Data Advance' each time — this is called clocking or strobing) to the register, this is the result. The left hand column corresponds to the left-most flip-flop's output pin, and so on.

So the serial output of the entire register is 10110000 . As you can see if we were to continue to input data, we would get exactly what was put in, but offset by four 'Data Advance' cycles. This arrangement is the hardware equivalent of a queue. Also, at any time, the whole register can be set to zero by bringing the reset (R) pins high.

This arrangement performs *destructive readout* - each datum is lost once it has been shifted out of the right-most bit.

Serial-in, parallel-out (SIPO)

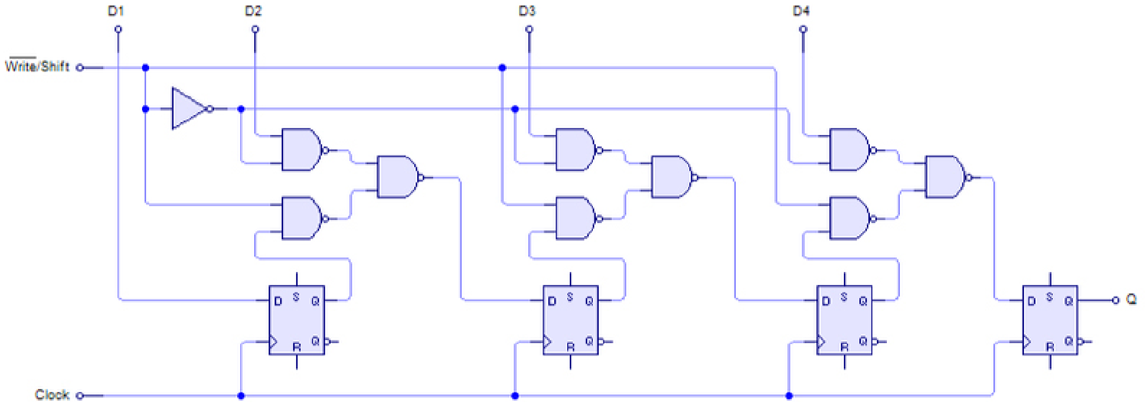
This configuration allows conversion from serial to parallel format. Data is input serially, as described in the SISO section above. Once the data has been input, it may be either read off at each output simultaneously, or it can be shifted out and replaced.



4-Bit SIPO Shift Register

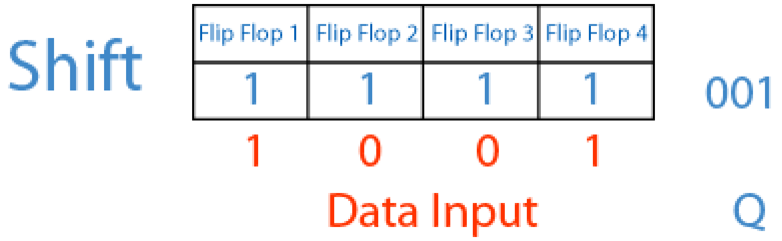
Parallel-in, serial-out (PISO)

This configuration has the data input on lines D1 through D4 in parallel format. To write the data to the register, the Write/Shift control line must be held LOW. To shift the data, the W/S control line is brought HIGH and the registers are clocked. The arrangement now acts as a PISO shift register, with D1 as the Data Input. However, as long as the number of clock cycles is not more than the length of the data-string, the Data Output, Q, will be the parallel data read off in order.



4-Bit PISO Shift Register

The image below shows the write/shift sequence, including the internal state of the shift register.


Shift
001
1 0 0 1
Data Input
Q

Flip Flop 1	Flip Flop 2	Flip Flop 3	Flip Flop 4
1	1	1	1

Uses

One of the most common uses of a shift register is to convert between serial and parallel interfaces. This is useful as many circuits work on groups of bits in parallel, but serial interfaces are simpler to construct. Shift registers can be used as simple delay circuits. Several bidirectional shift registers could also be connected in parallel for a hardware implementation of a stack.

SIPO registers are commonly attached to the output of microprocessors when more output pins are required than are available. This allows several binary devices to be controlled using only two or three pins - the devices in question are attached to the parallel outputs of the shift register, then the desired state of all those devices can be sent out of the microprocessor using a single serial connection. Similarly, PISO configurations are commonly used to add more binary inputs to a microprocessor than are available - each binary input (i.e. a switch or button, or more complicated circuitry designed to output high when active) is attached to a parallel input of the shift register, then the data is sent back via serial to the microprocessor using several fewer lines than originally required.

Shift registers can be used also as pulse extenders. Compared to monostable multivibrators, the timing has no dependency on component values, however it requires external clock and the timing accuracy is limited by a granularity of this clock. Example: Ronja Twister, where five 74164 shift registers create the core of the timing logic this way (schematic).

In early computers, shift registers were used to handle data processing: two numbers to be added were stored in two shift registers and clocked out into an arithmetic and logic unit (ALU) with the result being fed back to the input of one of the shift registers (the accumulator) which was one bit longer since binary addition can only result in an answer that is the same size or one bit longer.

Many computer languages include instructions to 'shift right' and 'shift left' the data in a register, effectively dividing by two or multiplying by two for each place shifted.

Very large serial-in serial-out shift registers (thousands of bits in size) were used in a similar manner to the earlier delay line memory in some devices built in the early 1970s. Such memories were sometimes called *circulating memory*. For example, the DataPoint 3300 terminal stored its display of 25 rows of 72 columns of upper-case characters using fifty-four 200-bit shift registers, arranged in six tracks of nine packs each, providing storage for 1800 six-bit characters. The shift register design meant that scrolling the terminal display could be accomplished by simply pausing the display output to skip one line of characters.

History

One of the first known examples of a shift register was in the Colossus, a code-breaking machine of the 1940s. It was a five-stage device built of vacuum tubes and thyratrons.

Chapter-8

Word (Computing)

In computing, **word** is a term for the natural unit of data used by a particular computer design. A word is simply a fixed sized group of bits that are handled together by the system. The number of bits in a word (the **word size** or **word length**) is an important characteristic of computer architecture.

The size of a word is reflected in many aspects of a computer's structure and operation; the majority of the registers in the computer are usually word sized and the amount of data transferred between the processing part computer and the memory system, in a single operation, is most often a word. The largest possible address size, used to designate a location in memory, is typically a hardware word (in other words, the full-sized natural word of the processor, as opposed to any other definition used on the platform).

Modern computers usually have a word size of 16, 32 or 64 bits but many other sizes have been used, including 8, 9, 12, 18, 24, 36, 39, 40, 48 and 60 bits. The slab is an example of a system with an earlier word size. Several of the earliest computers used the decimal base rather than binary, typically having a word size of 10 or 12 decimal digits, and some early computers had no fixed word length at all.

The size of a word can sometimes differ from the expected due to backward compatibility with earlier computers. If multiple compatible variations or a family of processors share a common architecture and instruction set but differ in their word sizes, their documentation and software may become notationally complex to accommodate the difference.

Uses of words

Depending on how a computer is organized, units of the word size may be used for:

- **Integer numbers**: Holders for integer numerical values may be available in one or in several different sizes, but one of the sizes available will almost always be the word. The other sizes, if any, are likely to be multiples or fractions of the word size. The smaller sizes are normally used only for efficient use of memory; when loaded into the processor, their values usually go into a larger, word sized holder.

- **Floating point numbers**: Holders for floating point numerical values are typically either a word or a multiple of a word.

- **Addresses**: Holders for memory addresses must be of a size capable of expressing the needed range of values but not be excessively large, so often the size used is the word though it can also be a multiple or fraction of the word size.

- **Registers**: Processor registers are designed with a size appropriate for the type of data they hold, e.g. integers, floating point numbers or addresses. Many computer architectures use "general purpose" registers that can hold any of several types of data, these registers must be sized to hold the largest of the types, historically this is the word size of the architecture though increasingly special purpose, larger, registers have been added to deal with newer types.

- **Memory-processor transfer**: When the processor reads from the memory subsystem into a register or writes a register's value to memory, the amount of data transferred is often a word. In simple memory subsystems, the word is transferred over the memory data bus, which typically has a width of a word or half word. In memory subsystems that use caches, the word-sized transfer is the one between the processor and the first level of cache; at lower levels of the memory hierarchy larger transfers (which are a multiple of the word size) are normally used.

- **Unit of address resolution**: In a given architecture, successive address values designate successive units of memory; this unit is the unit of address resolution. In most computers, the unit is either a character (e.g. a byte) or a word. (A few computers have used bit resolution.) If the unit is a word, then a larger amount of memory can be accessed using an address of a given size. On the other hand, if the unit is a byte, then individual characters can be addressed (i.e. selected during the memory operation).

- **Instructions**: Machine instructions are normally fractions or multiples of the architecture's word size. This is a natural choice since instructions and data usually share the same memory subsystem. In Harvard architectures the word sizes of instructions and data need not be related.

Word size choice

When a computer architecture is designed, the choice of a word size is of substantial importance. There are design considerations which encourage particular bit-group sizes for particular uses (e.g. for addresses), and these considerations point to different sizes for different uses. However, considerations of economy in design strongly push for one size, or a very few sizes related by multiples or fractions (submultiples) to a primary size. That preferred size becomes the word size of the architecture.

Character size is one of the influences on a choice of word size. Before the mid-1960s, characters were most often stored in six bits; this allowed no more than 64 characters, so alphabetics were limited to upper case. Since it is efficient in time and space to have the word size be a multiple of the character size, word sizes in this period were usually multiples of 6 bits (in binary machines). A common choice then was the 36-bit word, which is also a good size for the numeric properties of a floating point format.

After the introduction of the IBM System/360 design which used eight-bit characters and supported lower-case letters, the standard size of a character (or more accurately, a byte) became eight bits. Word sizes thereafter were naturally multiples of eight bits, with 16, 32, and 64 bits being commonly used.

Variable word architectures

Early machine designs included some that used what is often termed a *variable word length*. In this type of organization, a numeric operand had no fixed length but rather its end was detected when a character with a special marking was encountered. Such machines often used binary coded decimal for numbers. This class of machines included the IBM 702, IBM 705, IBM 7080, IBM 7010, UNIVAC 1050, IBM 1401, and IBM 1620.

Most of these machines work on one unit of memory at a time and since each instruction or datum is several units long, each instruction takes several cycles just to access memory. These machines are often quite slow because of this. For example, instruction fetches on an IBM 1620 Model I take 8 cycles just to read the 12 digits of the instruction (the Model II reduced this to 6 cycles, but reduced the fetch times to 4 cycles if both address fields were not needed by the instruction). Instruction execution took a completely variable number of cycles, depending on the size of the operands.

Word and byte addressing

The memory model of an architecture is strongly influenced by the word size. In particular, the resolution of a memory address, that is, the smallest unit that can be designated by an address, has often been chosen to be the word. In this approach, address values which differ by one designate adjacent memory words. This is natural in machines which deal almost always in word (or multiple-word) units, and has the advantage of allowing instructions to use minimally-sized fields to contain addresses, which can permit a smaller instruction size or a larger variety of instructions.

When byte processing is to be a significant part of the workload, it is usually more advantageous to use the byte, rather than the word, as the unit of address resolution. This allows an arbitrary character within a character string to be addressed straightforwardly. A word can still be addressed, but the address to be used requires a few more bits than the word-resolution alternative. The word size needs to be an integral multiple of the character size in this organization. This addressing approach was used in the IBM 360, and has been the most common approach in machines designed since then.

The power of two

Different amounts of memory are used to store data values with different degrees of precision. The commonly used sizes are usually a power of two multiple of the unit of address resolution (byte or word). Converting the index of an item in an array into the address of the item then requires only a shift operation rather than a multiplication. In some cases this relationship can also avoid the use of division operations. As a result, most modern computer designs have word sizes (and other operand sizes) that are a power of two times the size of a byte.

Size families

As computer designs have grown more complex, the central importance of a single word size to an architecture has decreased. Although more capable hardware can use a wider variety of sizes of data, market forces exert pressure to maintain backward compatibility while extending processor capability. As a result, what might have been the central word size in a fresh design has to coexist as an alternative size to the original word size in a backward compatible design. The original word size remains available in future designs, forming the basis of a size family.

In the mid-1970s, DEC designed the VAX to be a successor of the PDP-11. They used "word" for a 16-bit quantity while they used the term "longword" to refer to a 32-bit quantity. This is in contrast to earlier machines, where the natural unit of addressing memory would be called a *word*, while a quantity that is one half a word would be called, if anything, a *halfword*. In fitting with this scheme, a VAX "quadword" is 64 bits.

Another example is the x86 family, of which processors of three different word lengths (16-bit, later 32- and 64-bit) have been released. This leads to a common source of confusion because software is routinely ported from one word-length to the next. As a result, some APIs and documentation define or refer to an older (and thus shorter) word-length than software may be compiled for. For example, Microsoft's Windows API maintains the programming language definition of **WORD** as 16 bits, despite the fact that the API may be used on a 32- or 64-bit x86 processor, where the word size would be 32 or 64 bits, respectively. Data structures containing such 32- or 64-bit words are referred to as **DWORD** and **QWORD** respectively. A similar phenomenon has developed in Intel's x86 assembly language – because of backward compatibility in the instruction set, some instruction mnemonics carry "d" or "q" identifiers denoting "double-", "quad-" or "double-quad-", which are in terms of the architecture's original 16-bit word size.

Table of word sizes

key: **b: bits, d: decimal digits, *w*: word size of architecture, *n*: variable size**

Year	Computer Architecture	Word Size *w*	Integer Sizes	Floating Point Sizes	Instruction Sizes	Unit of Address Resolution	Char Size
1837	Babbage Analytical engine	50 d	*w*	—	5 different cards were used for different functions, exact size of cards not known	*w*	—
1941	Zuse Z3	22 b	—	*w*	8 b	*w*	—
1942	ABC	50 b	*w*	—	—	—	—
1944	Harvard Mark I	23 d	*w*	—	24 b	—	—
1946 (1948) {1953}	ENIAC (w/ Panel #16) {w/ Panel #26}	10 d	*w*, 2*w* (*w*) {*w*}	—	— (2*d*, 4*d*, 6*d*, 8*d*) {2*d*, 4*d*, 6*d*, 8*d*}	— — {*w*}	—
1951	UNIVAC I	12 d	*w*	—	½*w*	*w*	1 d
1952	IAS machine	40 b	*w*	—	½*w*	*w*	5 b
1952	Fast Universal Digital Computer M-2	34 b	*w?*	*w*	34 b = 4 b opcode plus 3× 10b address	10 b	—
1952	IBM 701	36 b	½*w*, *w*	—	½*w*	½*w*, *w*	6 b
1952	UNIVAC 60	*n* d	1*d*, ... 10*d*	—	—	—	2*d*, 3*d*
1953	IBM 702	*n* d	0*d*, ... 511*d*	—	5*d*	*d*	1 d
1953	UNIVAC 120	*n* d	1*d*, ... 10*d*	—	—	—	2*d*, 3*d*
1954 (1955)	IBM 650 (w/IBM 653)	10 d	*w*	— (*w*)	*w*	*w*	2 d
1954	IBM 704	36 b	*w*	*w*	*w*	*w*	6 b
1954	IBM 705	*n* d	0*d*, ... 255*d*	—	5*d*	*d*	1 d
1954	IBM NORC	16 d	*w*	*w*, 2*w*	*w*	*w*	—
1956	IBM 305	*n* d	1*d*, ... 100*d*	—	10*d*	*d*	1 d
1957	Autonetics Recomp I	40 b	*w*, 79 b, 8*d*, 15*d*	—	½*w*	½*w*, *w*	5 b

1958	UNIVAC II	12 d	*w*	—	½*w*	*w*	1 d
1958	SAGE	32 b	½*w*	—	*w*	*w*	6 b
1958	Autonetics Recomp II	40 b	*w*, 79 b, 8*d*, 15*d*	2*w*	½*w*	½*w*, *w*	5 b
1959	IBM 1401	*n* d	1*d*, ...	—	*d*, 2*d*, 4*d*, 5*d*, 7*d*, 8*d*	*d*	1 d
1959 (TBD)	IBM 1620	*n* d	2*d*, ...	— (4*d*, ... 102*d*)	12*d*	*d*	2 d
1960	LARC	12 d	*w*, 2*w*	*w*, 2*w*	*w*	*w*	2 d
1960	CDC 1604	48 b	*w*	*w*	½*w*	*w*	6 b
1960	IBM 1410	*n* d	1*d*, ...	—	*d*, 2*d*, 6*d*, 7*d*, 11*d*, 12*d*	*d*	1 d
1960	IBM 7070	10 d	*w*	*w*	*w*	*w*, *d*	2 d
1960	PDP-1	18 b	*w*	—	*w*	*w*	6 b
1961	IBM 7030 (Stretch)	64 b	1*b*, ... 64*b*, 1*d*, ... 16*d*	*w*	½*w*, *w*	b, ½*w*, *w*	1 b, ... 8 b
1961	IBM 7080	*n* d	0*d*, ... 255*d*	—	5*d*	*d*	1 d
1962	UNIVAC III	25 b, 6 d	*w*, 2*w*, 3*w*, 4*w*	—	*w*	*w*	6 b
1962	Autonetics D-17B Minuteman I Guidance Computer	27 b	11 b, 24 b	—	24 b	*w*	—
1962	UNIVAC 1107	36 b	⅙*w*, ⅓*w*, ½*w*, *w*	*w*	*w*	*w*	6 b
1962	IBM 7010	*n* d	1*d*, ...	—	*d*, 2*d*, 6*d*, 7*d*, 11*d*, 12*d*	*d*	1 d
1962	IBM 7094	36 b	*w*	*w*, 2*w*	*w*	*w*	6 b
1963	Gemini Guidance Computer	39 b	26 b	—	13 b	13 b, 26 b	—
1963 (1966)	Apollo Guidance Computer	15 b	*w*	—	*w*, 2*w*	*w*	—
1963	Saturn Launch Vehicle	26 b	*w*	—	13 b	*w*	—

	Digital Computer						
1964	CDC 6600	60 b	*w*	*w*	¼*w*, ½*w*	*w*	6 b
1964	Autonetics D-37C Minuteman II Guidance Computer	27 b	11 b, 24 b	—	24 b	*w*	4 b, 5 b
1965	IBM 360	32 b	½*w*, *w*, 1*d*, ... 16*d*	*w*, 2*w*	½*w*, *w*, 1½*w*	8 b	8 b
1965	UNIVAC 1108	36 b	⅙*w*, ¼*w*, ⅓*w*, ½*w*, *w*, 2*w*	*w*, 2*w*	*w*	*w*	6 b, 9 b
1965	PDP-8	12 b	*w*	—	*w*	*w*	8 b
1970	PDP-11	16 b	*w*	2*w*, 4*w*	*w*, 2*w*, 3*w*	8 b	8 b
1971	Intel 4004	4 b	*w*, *d*	—	2*w*, 4*w*	*w*	—
1972	Intel 8008	8 b	*w*, 2*d*	—	*w*, 2*w*, 3*w*	*w*	8 b
1972	Calcomp 900	9 b	*w*	—	*w*, 2*w*	*w*	8 b
1974	Intel 8080	8 b	*w*, 2*w*, 2*d*	—	*w*, 2*w*, 3*w*	*w*	8 b
1975	ILLIAC IV	64 b	*w*	*w*, ½*w*	*w*	*w*	—
1975	Motorola 6800	8 b	*w*, 2*d*	—	*w*, 2*w*, 3*w*	*w*	8 b
1975	MOS Tech. 6501 MOS Tech. 6502	8 b	*w*, 2*d*	—	*w*, 2*w*, 3*w*	*w*	8 b
1976	Cray-1	64 b	24 b, *w*	*w*	¼*w*, ½*w*	*w*	8 b
1976	Zilog Z80	8 b	*w*, 2*w*, 2*d*	—	*w*, 2*w*, 3*w*, 4*w*, 5*w*	*w*	8 b
1978 (1980)	Intel 8086 (w/Intel 8087)	16 b	½*w*, *w*, 2*d* (*w*, 2*w*, 4*w*)	— (2*w*, 4*w*, 5*w*, 17*d*)	½*w*, *w*, ... 7*w*	8 b	8 b
1978	VAX-11/780	32 b	¼*w*, ½*w*, *w*, 1*d*, ... 31*d*, 1*b*, ... 32*b*	*w*, 2*w*	¼*w*, ... 14¼*w*	8 b	8 b
1979	Motorola 68000	32 b	¼*w*, ½*w*, *w*, 2*d*	—	½*w*, *w*, ... 7½*w*	8 b	8 b
1982	Motorola	32 b	¼*w*, ½*w*,	—	½*w*, *w*, ... 7½*w*	8 b	8 b

(1983)	68020 (w/Motorola 68881)		*w*, 2*d*	(*w*, 2*w*, 2½*w*)			
1985	ARM1	32 b	*w*	—	*w*	8 b	8 b
1985	MIPS	32 b	¼*w*, ½*w*, *w*	*w*, 2*w*	*w*	8 b	8 b
1989	Intel 80486	16 b	½*w*, *w*, 2*d* *w*, 2*w*, 4*w*	2*w*, 4*w*, 5*w*, 17*d*	½*w*, *w*, ... 7*w*	8 b	8 b
1989	Motorola 68040	32 b	¼*w*, ½*w*, *w*, 2*d*	*w*, 2*w*, 2½*w*	½*w*, *w*, ... 7½*w*	8 b	8 b
1991	Alpha	64 b	¼*w*, ½*w*, *w*	*w*, 2*w*	½*w*	8 b	8 b
1991	Cray C90	64 b	32 b, *w*	*w*	¼*w*, ½*w*, 48b	*w*	8 b
1991	PowerPC	32–64 b	¼*w*, ½*w*, *w*	*w*, 2*w*	*w*	8 b	8 b
2000	IA-64	64 b	8 b, ¼*w*, ½*w*, *w*	½*w*, *w*	41 b	8 b	8 b
2002	XScale	32 b	*w*	*w*, 2*w*	½*w*, *w*	8 b	8 b

key: **b: bits, d: decimal digits, *w*: word size of architecture, *n*: variable size**

Chapter-9

Byte

The **byte**, is a unit of digital information in computing and telecommunications, that most commonly consists of eight bits. Historically, a byte was the number of bits used to encode a single character of text in a computer and it is for this reason the basic addressable element in many computer architectures.

The size of the byte has historically been hardware dependent and no definitive standards exist that mandate the size. The *de facto* standard of eight bits is a convenient power of two permitting the values 0 through 255 for one byte. Many types of applications use variables representable in eight or fewer bits, and processor designers optimize for this common usage. The popularity of major commercial computing architectures have aided in the ubiquitous acceptance of the 8-bit size. The term octet was defined to explicitly denote a sequence of 8 bits because of the ambiguity associated with the term byte.

History

The term *byte* was coined by Dr. Werner Buchholz in July 1956, during the early design phase for the IBM Stretch computer. It is a respelling of *bite* to avoid accidental mutation to *bit*.

Early computers were designed for 4-bit BCD code (binary coded decimal) or 6-bit code for printable "graphic set", which included 26 alphabetic characters (only uppercase), 10 Numerical digits, and from 11 to 25 special graphic symbols. To include the control characters and allow digital devices to communicate with each other and to process, store, and communicate character-oriented information such as written language, and lowercase characters, a 7-bit ASCII code was introduced. Since with just only one bit more an eight bits allows two four-bit patterns to efficiently encode two digits with binary coded decimal, the eight-bit EBCDIC character encoding was later adopted and promulgated as a standard by the IBM in the System/360, the preset byte.

The size of a byte was at first selected to be a multiple of existing teletypewriter codes, particularly the 6-bit codes used by the U.S. Army (Fieldata) and Navy.

In 1963, to end the use of incompatible teleprinter codes by different branches of the U.S. government, ASCII, a 7-bit code, was adopted as a Federal Information Processing Standard, making 6-bit bytes commercially obsolete. In the early 1960s, AT&T introduced digital telephony first on long-distance trunk lines. These used the 8-bit μ-law encoding. This large investment promised to reduce transmission costs for 8-bit data. The use of 8-bit codes for digital telephony also caused 8-bit data "octets" to be adopted as the basic data unit of the early Internet.

In the late 1970s, microprocessors such as the Intel 8008 (the direct predecessor of the 8080, and then the 8086 used in early PCs) could perform a small number of operations on four bits, such as the DAA (decimal adjust) instruction, and the *half carry* flag, which were used to implement decimal arithmetic routines. These four-bit quantities were called nibbles, in homage to the then-common 8-bit bytes.

Reasons for the ubiquity of the eight-bit byte include the popularity of the IBM System/360 architecture, introduced in the 1960s, and the 8-bit microprocessors, introduced in the 1970s.

The term octet is used to unambiguously specify a size of eight bits, and is used extensively in protocol definitions, for example.

Unit symbol

Prefixes for bit and byte multiples

Decimal		Binary		
Value	**SI**	**Value**	**IEC**	**JEDEC**
1000	k kilo	1024	Ki kibi	K kilo
1000^2	M mega	1024^2	Mi mebi	M mega
1000^3	G giga	1024^3	Gi gibi	G giga
1000^4	T tera	1024^4	Ti tebi	
1000^5	P peta	1024^5	Pi pebi	
1000^6	E exa	1024^6	Ei exbi	
1000^7	Z zetta	1024^7	Zi zebi	
1000^8	Y yotta	1024^8	Yi yobi	

The unit symbol for the byte is specified in IEEE 1541 and the Metric Interchange Format as the upper-case character B, while other standards, such as the International Electrotechnical Commission (IEC) standard IEC 60027, appear silent on the subject.

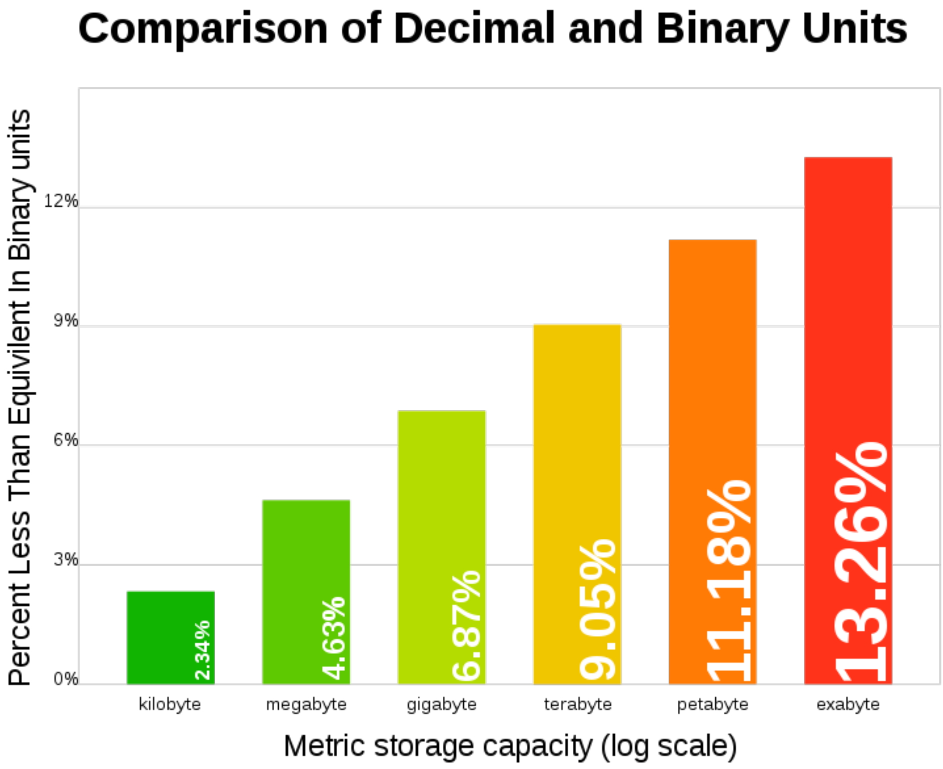
In the International System of Units (SI), B is the symbol of the bel, a unit of logarithmic power ratios named after Alexander Graham Bell. The usage of B for byte therefore conflicts with this definition. It is also not consistent with the SI convention that only units named after persons should be capitalized. However, there is little danger of confusion because the bel is a rarely used unit. It is used primarily in its decadic fraction, the decibel (dB), for signal strength and sound pressure level measurements, while a unit for one tenth of a byte, i.e. the decibyte, is never used.

The unit symbol kB is commonly used for kilobyte, but may be confused with the common meaning of kb for kilobit. IEEE 1541 specifies the lower case character b as the symbol for bit; however, the IEC 60027 and Metric-Interchange-Format specify bit (e.g., Mbit for megabit) for the symbol, a sufficient disambiguation from byte.

The lowercase letter o for octet is a commonly used symbol in several non-English languages (e.g., French and Romanian), and is also used with metric prefixes (for example, *ko* and *Mo*)

Today the harmonized ISO/IEC 80000-13:2008 – Quantities and units — Part 13: Information science and technology standard cancels and replaces subclauses 3.8 and 3.9 of IEC 60027-2:2005, namely those related to Information theory and Prefixes for binary multiples.

Unit multiples



Percentage difference between decimal and binary interpretations of the unit prefixes grows with increasing storage size.

There has been considerable confusion about the meanings of SI (or metric) prefixes used with the unit byte, especially concerning prefixes such as kilo (k or K) and mega (M) as shown in the chart *Prefixes for bit and byte*. Since computer memory is designed with binary logic, multiples are expressed in powers of 2, rather than 10. The software and computer industries often use binary estimates of the SI-prefixed quantities, while producers of computer storage devices prefer the SI values. This is the reason for specifying computer hard drive capacities of, say, 100 GB, when it contains 93 GiB of storage space.

While the numerical difference between the decimal and binary interpretations is small for the prefixes kilo and mega, it grows to over 20% for prefix yotta, illustrated in the linear-log graph (at right) of difference versus storage size.

Common uses

The byte is also defined as a data type in certain programming languages. The C and C++ programming languages, for example, define *byte* as an "*addressable unit of data large enough to hold any member of the basic character set of the execution environment*" (clause 3.6 of the C standard). The C standard requires that the `char` integral data type is capable of holding at least 255 different values, and is represented by at least 8 bits (clause 5.2.4.2.1). Various implementations of C and C++ reserve 8, 9, 16, 32, or 36 bits for the storage of a byte. The actual number of bits in a particular implementation is documented as `CHAR_BIT` as implemented in the `limits.h` file. Java's primitive `byte` data type is always defined as consisting of 8 bits and being a signed data type, holding values from −128 to 127.

In data transmission systems, a contiguous sequence of binary bits in a serial data stream, such as in modem or satellite communications, which is the smallest meaningful unit of data. These bytes might include start bits, stop bits, or parity bits, and thus could vary from 7 to 12 bits to contain a single 7-bit ASCII code.

Chapter-10

64-bit

Processors												
4-bit	8-bit	12-bit	16-bit	18-bit	24-bit	31-bit	32-bit	36-bit	48-bit	60-bit	**64-bit**	128-bit
Applications												
			16-bit				32-bit			**64-bit**		
Data Sizes												
nibble	octet	byte	word	dword	qword							

In computer architecture, **64-bit** integers, memory addresses, or other data units are those that are at most 64 bits (8 octets) wide. Also, 64-bit CPU and ALU architectures are those that are based on registers, address buses, or data buses of that size. **64-bit** is also a term given to a generation of computers in which 64-bit processors were the norm.

64-bit is a word size, that defines certain classes of computer architecture, buses, memory and CPUs, and by extension the software that runs on them. 64-bit CPUs have existed in supercomputers since the 1970s (Cray-1, 1975) and in RISC-based workstations and servers since the early 1990s. In 2003 they were introduced to the (previously 32-bit) mainstream personal computer arena, in the form of the x86-64 and 64-bit PowerPC processor architectures.

Without further qualification, a *64-bit* computer architecture generally has integer and addressing registers that are 64 bits wide, allowing direct support for 64-bit data types and addresses. However, a CPU might have external data buses or address buses with different sizes from the registers, even larger (the 32-bit Pentium had a 64-bit data bus, for instance). The term may also refer to the size of low-level data types, such as 64-bit floating-point numbers.

Architectural implications

Processor registers are typically divided into several groups: *integer*, *floating-point*, *SIMD*, *control*, and often special registers for address arithmetic which may have various uses and names such as *address*, *index* or *base registers*. However, in modern designs, these functions are often performed by more general purpose *integer* registers. In most processors, only integer and/or address-registers can be used to address data in memory, the other types cannot. The size of these registers therefore normally limits the amount of directly addressable memory, even if there are registers, such as floating-point registers, that are wider.

Most high performance 32-bit and 64-bit processors (some notable exceptions are most ARM and 32-bit MIPS CPUs) have integrated floating point hardware, which is often, but not always, based on 64-bit units of data. For example, although the x86/x87 architecture has instructions capable of loading and storing 64-bit (and 32-bit) floating-point values in memory, the internal data and register format is 80 bits wide. In contrast, the 64-bit Alpha family uses a 64-bit floating-point data and register format (as well as 64-bit integer registers).

History

Most CPUs are designed so that the contents of a single integer register can store the address (location) of any datum in the computer's virtual memory. Therefore, the total number of addresses in the virtual memory — the total amount of data the computer can keep in its working area — is determined by the width of these registers. Beginning in the 1960s with the IBM System/360 (which was an exception, in that it used the low order 24 bits of a word for addresses, resulting in a 16 MB [16 × 1024^2 bytes] address space size), then (amongst many others) the DEC VAX minicomputer in the 1970s, and then with the Intel 80386 in the mid-1980s, a *de facto* consensus developed that 32 bits was a convenient register size. A 32-bit address register meant that 2^{32} addresses, or 4 GB of RAM, could be referenced. At the time these architectures were devised, 4 GB of memory was so far beyond the typical quantities (4 MB) in installations that this was considered to be enough "headroom" for addressing. 4.29 billion addresses were considered an appropriate size to work with for another important reason: 4.29 billion integers are enough to assign unique references to most entities in applications like databases.

Some supercomputer architectures of the 1970s and 1980s used registers up to 64 bits wide. In the mid-1980s, Intel i860 development began culminating in a (too late for Windows NT) 1989 release. However, 32 bits remained the norm until the early 1990s, when the continual reductions in the cost of memory led to installations with quantities of RAM approaching 4 GB, and the use of virtual memory spaces exceeding the 4 GB ceiling became desirable for handling certain types of problems. In response, MIPS and DEC developed 64-bit microprocessor architectures, initially for high-end workstation and server machines. By the mid-1990s, HAL Computer Systems, Sun Microsystems, IBM, Silicon Graphics, and Hewlett Packard had developed 64-bit architectures for their workstation and server systems. A notable exception to this trend were mainframes from IBM, which then used 32-bit data and 31-bit address sizes; the IBM mainframes did not include 64-bit processors until 2000. During the 1990s, several low-cost 64-bit microprocessors were used in consumer electronics and embedded applications. Notably, the Nintendo 64 and the PlayStation 2 had 64-bit microprocessors before their introduction in personal computers. High-end printers and network equipment, as well as industrial computers also used 64-bit microprocessors such as the Quantum Effect Devices R5000. 64-bit computing started to drift down to the personal computer desktop from 2003 onwards, when some models in Apple's Macintosh lines switched to PowerPC 970 processors (termed "G5" by Apple) and the launch of AMD's 64-bit x86-64 extension to the x86 architecture, itself a response to Intel's Itanium gaining early operating systems support.

Limitations

Most 64-bit microprocessors on the market today have an artificial limit on the amount of memory they can address, considerably lower than what might be expected from 64 bits. For example, the AMD64 architecture currently has a 52 bit limit on physical memory and only supports a 48-bit virtual address space. This is 4 PB (4×1024^5 bytes) and 256 TB (256×1024^4 bytes), respectively. A PC cannot contain 4 petabytes of memory (due to the size of current memory chips if nothing else) but AMD envisioned large servers, shared memory clusters, and other uses of physical address space that might approach this in the foreseeable future, and the 52 bit physical address provides ample room for expansion while not incurring the cost of implementing 64-bit physical addresses. Similarly, the 48-bit virtual address space was designed to provide more than 65,000 times the 32 bit limit of 4 GB (4×1024^3 bytes), allowing ample room for expansion in the near future without incurring the overhead of translating full 64-bit addresses.

64-bit processor timeline

1961

IBM delivers the IBM 7030 Stretch supercomputer, which uses 64-bit data words and 32- or 64-bit instruction words.

1974

Control Data Corporation launches the CDC Star-100 vector supercomputer, which uses a 64-bit word architecture (previous CDC systems were based on a 60-bit architecture).

International Computers Limited launches the ICL 2900 Series with 32-bit, 64-bit, and 128-bit two's complement integers; 64-bit and 128-bit floating point; 32-bit, 64-bit and 128-bit packed decimal and a 128-bit accumulator register. The architecture has survived through a succession of ICL and Fujitsu machines. The latest is the Fujitsu Supernova, which emulates the original environment on 64-bit Intel processors.

1976

Cray Research delivers the first Cray-1 supercomputer, which is based on a 64-bit word architecture and will form the basis for later Cray vector supercomputers.

1983

Elxsi launches the Elxsi 6400 parallel minisupercomputer. The Elxsi architecture has 64-bit data registers but a 32-bit address space.

1989

Intel introduces the Intel i860 RISC processor. Marketed as a "64-Bit Microprocessor", it had essentially a 32-bit architecture, enhanced with a 3D Graphics Unit capable of 64-bit integer operations.

1991

MIPS Technologies produces the first 64-bit microprocessor, the R4000, which implements the MIPS III ISA, the third revision of their MIPS architecture. The CPU is used in SGI graphics workstations starting with the IRIS Crimson. Kendall Square Research deliver their first KSR1 supercomputer, based on a proprietary 64-bit RISC processor architecture running OSF/1.

1992

Digital Equipment Corporation (DEC) introduces the pure 64-bit Alpha architecture which was born from the PRISM project.

1993

Atari introduces the Atari Jaguar video game console, which includes some 64-bit wide data paths in its architecture.

1994

Intel announces plans for the 64-bit IA-64 architecture (jointly developed with Hewlett-Packard) as a successor to its 32-bit IA-32 processors. A 1998 to 1999 launch date is targeted.

1995

Sun launches a 64-bit SPARC processor, the UltraSPARC. Fujitsu-owned HAL Computer Systems launches workstations based on a 64-bit CPU, HAL's independently designed first-generation SPARC64. IBM releases the A10 and A30 microprocessors, 64-bit PowerPC AS processors. IBM also releases a 64-bit AS/400 system upgrade, which can convert the operating system, database and applications.

1996

Nintendo introduces the Nintendo 64 video game console, built around a low-cost variant of the MIPS R4000. HP releases an implementation of the 64-bit 2.0 version of their PA-RISC processor architecture, the PA-8000.

1997

IBM releases the RS64 line of 64-bit PowerPC/PowerPC AS processors.

1998

IBM releases the POWER3 line of full-64-bit PowerPC/POWER processors.

1999

Intel releases the instruction set for the IA-64 architecture. AMD publicly discloses its set of 64-bit extensions to IA-32, called x86-64 (later branded AMD64).

2000

IBM ships its first 64-bit z/Architecture mainframe, the zSeries z900. z/Architecture is a 64-bit version of the 32-bit ESA/390 architecture, a descendant of the 32-bit System/360 architecture.

2001

Intel finally ships its IA-64 processor line, after repeated delays in getting to market. Now branded Itanium and targeting high-end servers, sales fail to meet expectations.

2003

AMD introduces its Opteron and Athlon 64 processor lines, based on its AMD64 architecture which is the first x86-based 64-bit processor architecture. Apple also ships the 64-bit "G5" PowerPC 970 CPU produced by IBM. Intel maintains that its Itanium chips would remain its only 64-bit processors.

2004

Intel, reacting to the market success of AMD, admits it has been developing a clone of the AMD64 extensions named IA-32e (later renamed EM64T, then yet again renamed to Intel 64). Intel ships updated versions of its Xeon and Pentium 4 processor families supporting the new 64-bit instruction set.

VIA Technologies announces the Isaiah 64-bit processor.

2006

Sony, IBM, and Toshiba begin manufacturing of the 64-bit Cell processor for use in the PlayStation 3, servers, workstations, and other appliances.

64-bit operating system timeline

1985

Cray releases UNICOS, the first 64-bit implementation of the Unix operating system.

1993

DEC releases the 64-bit DEC OSF/1 AXP Unix-like operating system (later renamed Tru64 UNIX) for its systems based on the Alpha architecture.

1994

Support for the MIPS R8000 processor is added by Silicon Graphics to the IRIX operating system in release 6.0.

1995

DEC releases OpenVMS 7.0, the first full 64-bit version of OpenVMS for Alpha.

1996

Support for the MIPS R4000 processor is added by Silicon Graphics to the IRIX operating system in release 6.2.

1998

Sun releases Solaris 7, with full 64-bit UltraSPARC support.

2000

IBM releases z/OS, a 64-bit operating system descended from MVS, for the new zSeries 64-bit mainframes; 64-bit Linux on zSeries follows the CPU release almost immediately.

2001

NetBSD is the first operating system to run on the Intel Itanium processor at the processor's release. Further, Microsoft releases Windows XP 64-Bit Edition, also for the Itanium's IA-64 architecture, although it was able to run 32-bit applications through an execution layer.

2003

Apple releases its Mac OS X 10.3 "Panther" operating system which adds support for native 64-bit integer arithmetic on PowerPC 970 processors. Several Linux distributions release with support for AMD64. Microsoft announces plans to create a version of its Windows operating system to support the AMD64 architecture, with backwards compatibility with 32-bit applications. FreeBSD releases with support for AMD64.

2005

On January 31, Sun releases Solaris 10 with support for AMD64 and EM64T processors. On April 29, Apple releases Mac OS X 10.4 "Tiger" which provides limited support for 64-bit command-line applications on machines with PowerPC 970 processors; later versions for Intel-based Macs supported 64-bit command-line applications on Macs with EM64T processors. On April 30, Microsoft releases Windows XP Professional x64 Edition for AMD64 and EM64T processors.

2006

Microsoft releases Windows Vista, including a 64-bit version for AMD64/EM64T processors that retains 32-bit compatibility. In the 64-bit version, all Windows applications and components are 64-bit, although many also have their 32-bit versions included for compatibility with plugins.

2007

Apple releases Mac OS X 10.5 "Leopard", which fully supports 64-bit applications on machines with PowerPC 970 or EM64T processors.

2009

Apple releases Mac OS X 10.6, "Snow Leopard," which ships with a 64-bit kernel for AMD64/Intel64 processors, although only certain recent models of Apple computers will run the 64-bit kernel by default. Most applications bundled with Mac OS X 10.6 are now also 64-bit. Microsoft releases Windows 7, which, like Windows Vista, includes a full 64-bit version for AMD64/Intel 64 processors; most new computers are loaded by default with a 64-bit version. It also releases Windows Server 2008 R2, which is the first 64-bit only operating system released by Microsoft.

32-bit vs 64-bit

A change from a 32-bit to a 64-bit architecture is a fundamental alteration, as most operating systems must be extensively modified to take advantage of the new architecture, because that software has to manage the actual memory addressing hardware. Other software must also be ported to use the new capabilities; older software is usually supported through either a *hardware compatibility mode* (in which the new processors support the older 32-bit version of the instruction set as well as the 64-bit version), through software emulation, or by the actual implementation of a 32-bit processor core within the 64-bit processor (as with the Itanium processors from Intel, which include an x86 processor core to run 32-bit x86 applications). The operating systems for those 64-bit architectures generally support both 32-bit and 64-bit applications.

One significant exception to this is the AS/400, whose software runs on a virtual ISA, called TIMI (Technology Independent Machine Interface) which is translated to native machine code by low-level software before being executed. The low-level software is all that has to be rewritten to move the entire OS and all software to a new platform, such as when IBM transitioned their line from the older 32/48-bit "IMPI" instruction set to 64-bit PowerPC (IMPI wasn't anything like 32-bit PowerPC, so this was an even bigger transition than from a 32-bit version of an instruction set to a 64-bit version of the same instruction set).

While 64-bit architectures indisputably make working with large data sets in applications such as digital video, scientific computing, and large databases easier, there has been considerable debate as to whether they or their 32-bit compatibility modes will be faster than comparably-priced 32-bit systems for other tasks. In x86-64 architecture (AMD64), the majority of the 32-bit operating systems and applications are able to run smoothly on the 64-bit hardware.

A compiled Java program can run on a 32 bit or 64 bit Java virtual machine without modification. The lengths and precision of all the built in types are specified by the standard and are not dependent on the underlying architecture. Java programs that run on a 64 bit Java virtual machine have access to a larger address space.

Speed is not the only factor to consider in a comparison of 32-bit and 64-bit processors. Applications such as multi-tasking, stress testing, and clustering—for HPC (high-performance computing)—may be more suited to a 64-bit architecture when deployed appropriately. 64-bit clusters have been widely deployed in large organizations such as IBM, HP and Microsoft, for this reason.

Pros and cons

A common misconception is that 64-bit architectures are no better than 32-bit architectures unless the computer has more than 4 GB of main memory. This is not entirely true:

- Some operating systems and certain hardware configurations limit the physical memory space to 3 GB on IA-32 systems, due to much of the 3–4 GB region being reserved for hardware addressing. This is not present in 64-bit architectures, which can use 4 GB of memory and more. However, IA-32 processors from the Pentium II onwards allow for a 36-bit *physical* memory address space, using Physical Address Extension (PAE), which gives a 64 GB physical address range, of which up to 62 GB may be used by main memory; operating systems that support PAE may not be limited to 4GB of physical memory, even on IA-32 processors.

- Some operating systems reserve portions of process address space for OS use, effectively reducing the total address space available for mapping memory for user programs. For instance, Windows XP DLLs and other user mode OS components are mapped into each process's address space, leaving only 2 to 3 GB (depending on the settings) address space available. This limit is currently much higher on 64-bit operating systems and does not realistically restrict memory usage.

- Memory-mapped files are becoming more difficult to implement in 32-bit architectures. A 4 GB file is no longer uncommon, and such large files cannot be memory mapped easily to 32-bit architectures; only a region of the file can be mapped into the address space, and to access such a file by memory mapping, those regions will have to be mapped into and out of the address space as needed. This is a problem, as memory mapping remains one of the most efficient disk-to-memory methods, when properly implemented by the OS.

- Some programs such as encoders, decoders and encryption software can benefit greatly from 64-bit registers (if the software is 64-bit compiled), while the performance of other programs, such as 3D graphics-oriented ones, remains unaffected when switching from a 32-bit environment to a 64-bit one. It is unusual for a 64-bit program to perform worse than its 32-bit equivalent and usually only happens due to a bug.

- Some 64-bit architectures, such as x86-64, allow for more general purpose registers than their 32-bit counterparts. This is a significant speed increase for tight loops since the processor doesn't have to fetch data from the cache or main memory if the data can fit in the available registers.

  Example in C:

```
for (a=0; a<100; a++)
{
  b = a;
  c = b;
  d = c;
  e = d;
}
```

If a processor only has the ability to keep two three values/variables in registers it would need to move some values between the stack and registers to be able to process variable d and e as well; this is a process that takes a lot of CPU cycles. A processor that is capable of holding all the values/variables in registers can simply just loop through this without needing to move data between registers and memory for each iteration. This behavior can easily be compared with virtual memory, although any effects are contingent upon the compiler.

The main disadvantage of 64-bit architectures is that relative to 32-bit architectures, the same data occupies more space in memory (due to swollen pointers and possibly other types and alignment padding). This increases the memory requirements of a given process and can have implications for efficient processor cache utilization. Maintaining a partial 32-bit model is one way to handle this and is in general reasonably effective. For example, the z/OS operating system takes this approach currently, requiring program code to reside in 31-bit address spaces (the high order bit is not used in address calculation on the underlying hardware platform) while data objects can optionally reside in 64-bit regions.

Currently, most proprietary x86 software is compiled into 32-bit code, with less being also compiled into 64-bit code (although the trend is rapidly equalizing), so much does not take advantage of the larger 64-bit address space or wider 64-bit registers and data paths on x86 processors, or the additional registers in 64-bit mode. However, users of most RISC platforms, and users of free or open source operating systems (where the source code is available for recompiling with a 64-bit compiler) have been able to use exclusive 64-bit computing environments for years. Not all such applications require a large address space nor manipulate 64-bit data items, so they wouldn't benefit from the larger address space or wider registers and data paths. The main advantage to 64-bit versions of such applications is the ability to access more registers in the x86-64 architecture.

Software availability

x86-based 64-bit systems sometimes lack equivalents to software that is written for 32-bit architectures. The most severe problem in Microsoft Windows is incompatible device drivers. Most 32-bit application software can run on a 64-bit operating system in a compatibility mode, also known as an emulation mode, e.g. Microsoft WoW64 Technology for IA-64 and AMD64. The 64-bit Windows Native Mode driver environment runs atop 64-bit NTDLL.DLL which cannot call 32-bit Win32 subsystem code (often devices whose actual hardware function is emulated in user mode software, like Winprinters). Because 64-bit drivers for most devices were not available until early 2007 (Vista x64), using 64-bit Microsoft Windows operating system was considered a challenge. However, the trend is changing towards 64-bit computing as most manufacturers provide both 32-bit and 64-bit drivers nowadays, so this issue is most likely to occur when attempting to use older peripherals.

This is less of a problem with open source drivers that are already available for a 32-bit OS, since they can be modified to be 64-bit compatible, if necessary. Furthermore, support for hardware made before early 2007 was equally troubling for opensource platforms due to their small market shares in desktop market.

On most Macs, Mac OS X runs with a 32-bit kernel even on 64-bit-capable processors, but the 32-bit kernel can run 64-bit user-mode code; this allows those Macs to support 64-bit processes while still supporting 32-bit device drivers - although not 64-bit drivers and performance advantages that would come with them. On systems with 64-bit processors, both the 32-bit and 64-bit Mac OS X kernel can run 32-bit user-mode code, and all versions of Mac OS X include 32-bit versions of libraries that 32-bit applications would use, so 32-bit user-mode software for Mac OS X will run on those systems.

Linux and most other Unix-like operating systems, and the C and C++ toolchains for them, have supported 64-bit processors for many years: releasing 64-bit versions of their operating system before official Microsoft releases. Many applications and libraries for those platforms are open source, written in C and C++, so that, if it's 64-bit-safe, they can be compiled into 64-bit versions. This source based distribution model with an emphasis on frequent releases and cutting edge code makes availability of application software for those operating systems less of an issue.

64-bit data models

Converting application software written in a high-level language from a 32-bit architecture to a 64-bit architecture varies in difficulty. One common recurring problem is that some programmers assume that pointers have the same length as some other data type. These programmers assume they can transfer quantities between these data types without losing information. Those assumptions happen to be true on some 32-bit machines (and even some 16-bit machines), but they are no longer true on 64-bit machines. This common mistake is often called "the heresy that 'all the world's a VAX.'". The C programming language and its descendant C++ make it particularly easy to make this sort of mistake. Differences between the C89 and C99 language standards also exacerbate the problem.

To avoid this mistake in C and C++, the `sizeof` operator can be used to determine the size of these primitive types if decisions based on their size need to be made, both at compile- and run-time. Also, the <limits.h> header in the C99 standard, and numeric_limits class in <limits> header in the C++ standard, give more helpful info; sizeof only returns the size in *chars*. This used to be misleading, because the standards leave the definition of the `CHAR_BIT` macro, and therefore the number of bits in a *char*, to the implementations. However, except for those compilers targeting DSPs, "64 bits == 8 chars of 8 bits each" has become the norm.

One needs to be careful to use the `ptrdiff_t` type (in the standard header `<stddef.h>`) for the result of subtracting two pointers; too much code incorrectly uses "int" or "long" instead. To represent a pointer (rather than a pointer difference) as an integer, use `uintptr_t` where available (it is only defined in C99, but some compilers otherwise conforming to an earlier version of the standard offer it as an extension).

Neither C nor C++ define the length of a pointer, int, or long to be a specific number of bits. In C99, however, stdint.h provides names for integer types with certain numbers of bits where those types are available.

Specific C-language data models

In most programming environments on 32-bit machines, pointers, "int" (that is, integer) types, and "long" (that is, long integer) types are all 32 bits wide.

However, in many programming environments on 64-bit machines, "int" variables are still 32 bits wide, but long integers and pointers are 64 bits wide. These are described as having an **LP64** data model. Another alternative is the **ILP64** data model in which all three data types are 64 bits wide, and even **SILP64** where "short" integers are also 64 bits wide. However, in most cases the modifications required are relatively minor and straightforward, and many well-written programs can simply be recompiled for the new environment without changes. Another alternative is the **LLP64** model, which maintains compatibility with 32-bit code by leaving both int and long as 32-bit. "LL" refers to the "long long integer" type, which is at least 64 bits on all platforms, including 32-bit environments.

64-bit data models

Data model	short (integer)	int	long (integer)	long long	pointers/size_t	Sample operating systems
LLP64/ IL32P64	16	32	32	64	64	Microsoft Windows (X64/IA-64)
LP64/ I32LP64	16	32	64	64	64	Most Unix and Unix-like systems, e.g. Solaris, Linux, and Mac OS X
ILP64	16	64	64	64	64	HAL Computer Systems port of Solaris to SPARC64
SILP64	64	64	64	64	64	Unicos

Many 64-bit compilers today use the **LP64** model (including Solaris, AIX, HP-UX, Linux, Mac OS X, FreeBSD, and IBM z/OS native compilers). Microsoft's Visual C++ compiler uses the **LLP64** model. The disadvantage of the LP64 model is that storing a long into an int may overflow. On the other hand, casting a pointer to a long will work. In the LLP model, the reverse is true. These are not problems which affect fully standard-compliant code but code is often written with implicit assumptions about the widths of integer types.

Note that a programming model is a choice made on a per-compiler basis, and several can coexist on the same OS. However, the programming model chosen as the primary model for the OS API typically dominates.

Another consideration is the data model used for drivers. Drivers make up the majority of the operating system code in most modern operating systems (although many may not be loaded when the operating system is running). Many drivers use pointers heavily to manipulate data, and in some cases have to load pointers of a certain size into the hardware they support for DMA. As an example, a driver for a 32-bit PCI device asking the device to DMA data into upper areas of a 64-bit machine's memory could not satisfy requests from the operating system to load data from the device to memory above the 4 gigabyte barrier, because the pointers for those addresses would not fit into the DMA registers of the device. This problem is solved by having the OS take the memory restrictions of the device into account when generating requests to drivers for DMA, or by using an IOMMU.

Current 64-bit microprocessor architectures

64-bit microprocessor architectures (as of January 2011) include:

- The 64-bit extension created by AMD to Intel's x86 architecture (later licensed by Intel); commonly known as "x86-64", "AMD64", or "x64":
    - AMD's AMD64 extensions (used in Athlon 64, Opteron, Sempron, Turion 64, Phenom, Athlon II and Phenom II processors)
    - Intel's Intel 64 extensions (used in newer Celeron, Pentium 4, Pentium D, and Xeon processors, in Core 2, Core i3, Core i5, and Core i7 processors, and in some Atom processors)
    - VIA Technologies' 64-bit extensions, used in the VIA Nano processors
- The 64-bit version of the Power Architecture:
    - IBM's POWER6 and POWER7 processors
    - IBM's PowerPC 970 processor
    - The Cell Broadband Engine used in the PlayStation 3, designed by IBM, Toshiba and Sony, combines a 64-bit Power architecture processor with seven or eight Synergistic Processing Elements.
    - IBM's "Xenon" processor used in the Microsoft Xbox 360 comprises three 64-bit PowerPC cores.
- SPARC V9 architecture:
    - Sun's UltraSPARC processors
    - Fujitsu's SPARC64 processors
- IBM's z/Architecture, used by IBM zSeries and System z9 mainframes, a 64-bit version of the ESA/390 architecture
- Intel's IA-64 architecture (used in Itanium processors)
- MIPS Technologies' MIPS64 architecture

Most 64-bit processor architectures that are derived from 32-bit processor architectures can execute code for the 32-bit version of the architecture natively without any performance penalty. This kind of support is commonly called *bi-arch support* or more generally *multi-arch support*.

Chapter-11

16-bit, 26-bit and 31-bit

16-bit

In computer architecture, **16-bit** integers, memory addresses, or other data units are those that are at most 16 bits (2 octets) wide. Also, 16-bit CPU and ALU architectures are those that are based on registers, address buses, or data buses of that size. **16-bit** is also a term given to a generation of computers in which 16-bit processors were the norm.

16-bit architecture

The HP BPC, introduced in 1975, was the world's first 16-bit microprocessor. Prominent 16-bit processors include the PDP-11, Intel 8086, Intel 80286 and the WDC 65C816. The Intel 8088 was program-compatible with the Intel 8086, and was 16-bit in that its registers were 16 bits long and arithmetic instructions, even though its external bus was 8 bits wide. Other notable 16-bit processors include the Texas Instruments TMS9900 and the Zilog Z8000.

A 16-bit integer can store 2^{16} (or 65,536) unique values. In an unsigned representation, these values are the integers between 0 and 65,535; using two's complement, possible values range from −32,768 to 32,767. Hence, a processor with 16-bit memory addresses can directly access 64 KB of byte-addressable memory.

16-bit processors have been almost entirely supplanted in the personal computer industry, but remain in use in a wide variety of embedded applications. For example the 16-bit XAP processor is used in many ASICs.

The 16/32-bit Motorola 68000 and Intel 386SX

The Motorola 68000 is sometimes called *16-bit* because its internal and external data buses were 16 bits wide, however it could be considered a 32-bit processor in that the general purpose registers were 32 bits wide and most arithmetic instructions supported

32-bit arithmetic. The MC68000 was a microcoded processor with three internal 16-bit ALU units. Only 24-bits of the Program Counter were available on original DIP packages, with up to 16 megabytes of addressable RAM. MC68000 software is 32-bit in nature, and forwards-compatible with other 32-bit processors. The MC68008 was a version of the 68000 with 8-bit external data path and 1 megabyte addressing. Several Apple Inc. Macintosh models; e.g., LC series, used 32-bit MC68020 and MC68030 processors on a 16-bit data bus to save cost.

Similar analysis applies to Intel's 80286 CPU replacement called the 386SX which is a 32-bit processor with 32-bit ALU and internal 32-bit data paths with a 16-bit external bus and 24-bit addressing of the processor it replaced.

The 68000 processor of the Sega Mega Drive/Genesis was a highly advertised feature of the video game system. Due to the saturation of this advertising, the 1988-1995 era (fourth generation) of video game consoles is often called *the 16-bit era*.

16-bit file format

A 16-bit file format is a binary file format for which each data element is defined on 16 bits (or 2 Bytes). An example of such a format is UTF-16 and the Windows Metafile Format.

16-bit memory models

Similar to 64-bit's data models, the 16-bit Intel architecture allows for different memory models—ways to access a particular memory location. The reason for using a specific memory model is the size of the assembler instructions or required storage for pointers. Compilers of the 16-bit era generally had the following type-width characteristic:

16-bit data model

Data model	`short`	`int`	`long`	Pointers
IP16L32 (near)	16	16	32	16
I16LP32 (far)	16	16	32	32

Tiny

Code and data will be in the same segment (especially, the registers CS,DS,ES,SS will point to the same segment); *near* pointers are always used. Code, data and stack together cannot exceed 64K.

Small

Code and data will be in different segments, and near pointers are always used. There will be 64K of space for code and 64K for data/stack.

Medium

Code pointers will use *far* pointers, enabling access to 1 MB. Data pointers remain to be of the near type.

Compact

Data pointers will use far and code will use near pointers.

Large/huge
Code and data pointers will be far.

26-bit

In computer architecture, **26-bit** integers, memory addresses, or other data units are those that are at most 26 bits wide. Also, 26-bit CPU and ALU architectures are those that are based on registers, address buses, or data buses of that size.

In the ARM processor architecture, **26-bit** refers to the design used in the original ARM processors, where the Program Counter (**PC**) and Processor Status Register (**PSR**) were combined into one 32-bit register (R15), the status flags filling the high 6 bits and the Program Counter taking up the lower 26 bits.

In fact, because the program counter is always word-aligned the lowest two bits are always zero which allowed the designers to reuse these two bits to hold the processor's mode bits too. The four modes allowed were USR26, SVC26, IRQ26, FIQ26; contrast this with the 32 possible modes available when the program status was separated from the program counter in more recent ARM architectures.

This design enabled more efficient program execution, as the Program Counter and status flags could be saved and restored with a single operation. This resulted in faster subroutine calls and interrupt response than traditional designs, which would have to do two register loads or saves when calling or returning from a subroutine.

History

Despite being 32-bit internally, processors prior to the ARM6 had only a 26-bit PC and address bus, and were consequently limited to 64 MB of addressable memory. This was still a vast amount of memory at the time, but because of this limitation, architectures since have included various steps away from the original 26-bit design.

The ARM6 introduced a 32-bit PC and separate PSR, as well as a 32-bit address bus, allowing 4 GB of memory to be addressed. The change in the PC/PSR layout caused incompatibility with code written for previous architectures, so the processor also included a 26-bit compatibility mode which used the old PC/PSR combination. The processor could still address 4 GB in this mode, but could not execute anything above address 3FFFFFC (64 MB). This mode was used by RISC OS running on the Acorn Risc PC to utilise the new processors while retaining compatibility with existing software.

More recent ARM architectures such as Intel's XScale have dropped the 26-bit mode altogether.

31-bit

In computer architecture, **31-bit** integers, memory addresses, or other data units are those that are at most 31 bits (32 bits minus 1 unused/reserved bit) wide. Also, 31-bit CPU and ALU architectures are those that are based on registers, address buses, or data buses of that size.

Perhaps the only computing architecture based on 31-bit addressing is one of computing's most famous and most profitable. In 1983, IBM introduced 31-bit addressing in the System/370-XA mainframe architecture as an upgrade to the 24-bit addressing of earlier models. This enhancement allowed address spaces to be 128 times larger, permitting programs to address memory above 16 MiB (referred to as "above the line").

In the System/360 and early System/370 architectures, addresses were always stored in 32-bit words, but the machines ignored the top 8 bits of the address resulting in 24-bit addressing. With the XA extension, no bits in the word were ignored.

The transition was tricky: assembly language programmers had been using the spare byte at the top of addresses for flags for almost twenty years. IBM chose to support two forms of addressing to minimize the pain: if the most significant bit (bit 0) of a 32-bit address was on, the next 31 bits were interpreted as the virtual address. If the most significant bit was off, then only the lower 24 bits were treated as the address (just as with pre-XA systems). Thus programs could continue using the seven low-order bits of the top byte for other purposes as long as they left the top bit off. The only programs requiring modification were those that set the top (leftmost) bit of a word containing an address. This also affected address comparisons: The leftmost bit of a word is also interpreted as a sign-bit, indicating a negative number if bit 0 is on. Programs that use signed arithmetic comparison instructions could get reversed results. Two equivalent addresses could be compared as non-equal if one of them had the sign bit turned on even if the remaining bits were identical. Fortunately, most of this was invisible to programmers using high-level languages like COBOL or FORTRAN, and IBM aided the transition with dual mode hardware for a period of time.

Certain machine instructions in this 31-bit architecture alter the addressing mode bit as a possibly intentional side effect. For example, the original subroutine call instruction BAL stores certain status information in the top byte of the return address. A BAS instruction was added to support 31-bit return addresses.

In the 1990s IBM introduced 370/ESA architecture (later named 390/ESA and finally ESA/390 or System/390, in short S/390), completing the evolution to full 31-bit virtual addressing and keeping this addressing mode flag. These later architectures support more than 2 GiB of physical memory and support multiple concurrent address spaces up to 2 GiB each in size. As of mid-2006 there still are not too many programs unduly constrained by this multiple 31-bit architecture.

Nonetheless, IBM broke the 2 GiB linear addressing barrier ("the bar") in 2000 with the introduction of the first 64-bit z/Architecture system, the IBM zSeries Model 900. Unlike the XA transition, z/Architecture does not reserve a top bit to identify earlier code. Yet z/Architecture does maintain compatibility with 24-bit and 31-bit code, even older code running concurrently with newer 64-bit code.

Since Linux/390 was first released for the existing 32-bit data/31-bit addressing hardware in 1999, initial mainframe Linux applications compiled in pre-z/Architecture mode are also limited to 31-bit addressing. This limitation disappeared with 64-bit hardware, 64-bit Linux on zSeries, and 64-bit Linux applications. The 64-bit Linux distributions still support 31-bit programs.

IBM's 31-bit architecture supports expanded storage, allowing 31-bit code to make use of additional memory. However, at any one instant, a maximum of 2 GiB is in each working address space. For 31-bit Linux it is possible to assign memory above the 2 GiB bar as a RAM disk.

Chapter-12

32-bit, 36-bit, 8-bit

32-bit

In computer architecture, **32-bit** integers, memory addresses, or other data units are those that are at most 32 bits (4 octets) wide. Also, 32-bit CPU and ALU architectures are those that are based on registers, address buses, or data buses of that size. **32-bit** is also a term given to a generation of computers in which 32-bit processors were the norm.

The range of integer values that can be stored in 32 bits is 0 through 4,294,967,295 or −2,147,483,648 through 2,147,483,647 using two's complement encoding. Hence, a processor with 32-bit memory addresses can directly access 4 GB of byte-addressable memory.

The external address and data buses are often wider than 32 bits but both of these are stored and manipulated internally in the processor as 32-bit quantities. For example, the Pentium Pro processor is a 32-bit machine, but the external address bus is 36 bits wide, and the external data bus is 64 bits wide.

Architecture

Prominent 32-bit instruction set architectures include the IBM System/360, the DEC VAX, the ARM, the MIPS, and the Intel IA-32.

Images

In digital images/pictures, 32-bit can refer to 24-bit truecolor images with an 8-bit alpha channel.

Alternatively it may refer to 32-bit per channel rather than 24-bit colour + 8-bit alpha. 32-bit per channel images are used to represent values brighter than white, these values can then be used to more accurately retain bright highlights when either lowering the exposure of the image or when it is seen through a dark filter or dull reflection.

An example of this would be the reflection seen in an oil slick; even though the reflection is only a fraction of that seen in a mirror surface, the reflection of highlights can still be seen as bright white areas, not dull grey shapes.

32-bit file format

A 32-bit file format is a binary file format for which each elementary information is defined on 32 bits (or 4 Bytes). An example of such a format is the Enhanced Metafile Format.

36-bit

In computer architecture, **36-bit** integers, memory addresses, or other data units are those that are at most 36 bits (6 characters) wide. Also, 36-bit CPU and ALU architectures are those that are based on registers, address buses, or data buses of that size.

Many early computers aimed at the scientific market had a 36-bit word length. This word length was just long enough to represent positive and negative integers to an accuracy of ten decimal digits (35 bits would have been the minimum). It also allowed the storage of six alphanumeric characters encoded in a six-bit character encoding. Prior to the introduction of computers, the state of the art in precision scientific and engineering calculation was the ten-digit, electrically powered, mechanical calculator, such as those manufactured by Friden, Marchant and Monroe. These calculators had a column of keys for each digit and operators were trained to use all their fingers when entering numbers, so while some specialized calculators had more columns, ten was a practical limit. Computers, as the new competitor, had to match that accuracy. Decimal computers sold in that era, such as the IBM 650 and the IBM 7070, had a word length of ten digits, as did ENIAC, one of the earliest computers.

Computers with 36-bit words included the MIT Lincoln Laboratory TX-2, the IBM 701/704/709/7090/7094, the UNIVAC 1103/1103A/1105/1100/2200, the General Electric GE-600/Honeywell 6000, the Digital Equipment Corporation PDP-6/PDP-10 (as used in the DECsystem-10/DECSYSTEM-20), and the Symbolics 3600 series. Smaller machines like the PDP-1/PDP-9/PDP-15 used 18-bit words, so a double word would be 36 bits. EDSAC had a similar scheme.

These computers used 18-bit word addressing, not byte addressing, giving an address space of 2^{18} 36-bit words, approximately 1 megabyte of storage. Many of them were originally limited to a similar amount of physical memory as well. Architectures that survived evolved over time to support larger virtual address spaces using memory segmentation or other mechanisms.

The common character packings included

- six 6-bit Fieldata or IBM BCD characters (ubiquitous in early usage)
- five 7-bit characters and 1 unused bit (the usual PDP-6/10 convention, called *five-seven ASCII*)
- four 8-bit characters (7-bit ASCII plus 1 unused bit or 8-bit EBCDIC) and 4 unused bits
- four 9-bit characters (the Multics convention).

Characters were extracted from words either using standard shift and mask operations or with special-purpose hardware supporting 6-bit, 9-bit, or variable-length characters. The Univac 1100/2200 used the *partial word designator* of the instruction, the "J" field, to access characters. The GE-600 used special indirect words to access 6- and 9-bit characters; the PDP-6/10 had special instructions to access arbitrary-length byte fields. The C programming language requires that all memory be accessible as bytes, so C implementations on 36-bit machines use 9-bit bytes.

By the time IBM introduced System/360, scientific calculations had shifted to floating point and mechanical calculators were no longer a competitor. The 360s also included instructions for variable length decimal arithmetic for commercial applications, so the practice of using word lengths that were a power of two quickly became universal.

8-bit

In computer architecture, **8-bit** integers, memory addresses, or other data units are those that are at most 8 bits (1 octet) wide. Also, 8-bit CPU and ALU architectures are those that are based on registers, address buses, or data buses of that size. **8-bit** is also a term given to a generation of computers in which 8-bit processors were the norm.

Eight-bit CPUs normally use an 8-bit data bus and a 16-bit address bus which means that their address space is limited to 64 KiB. This is not a "natural law", however, so there are exceptions.

The first widely adopted 8-bit microprocessor was the Intel 8080, being used in many hobbyist computers of the late 1970s and early 1980s, often running the CP/M operating system. The Zilog Z80 (compatible with the 8080) and the Motorola 6800 were also used in similar computers. The Z80 and the MOS Technology 6502 8-bit CPUs were widely used in home computers and game consoles of the '70s and '80s. Many 8-bit CPUs or microcontrollers are the basis of today's ubiquitous embedded systems.

There are 2^8 (256) possible values for 8 bits.

The first microprocessors had a 4-bit word length and were developed around 1970. The first commercial microprocessor was the BCD-based Intel 4004 (1971), developed for calculator applications. The first commercial 8-bit processor was the Intel 8008 (1972) which was originally intended for intelligent terminals. Most competitors to Intel started off with such character oriented 8-bit microprocessors. Modernized variants of these 8-bit machines are still one of the most common types of processor in embedded systems.

List of 8-bit CPUs

A CPU can be classified on the basis of the data it can access in a single operation. An 8-bit processor can access 8 bits of data in a single operation, as opposed to a 16-bit processor, which can access 16 bits of data in a single operation. 8 Bits *Examples of 8-bit processors (very incomplete):*

- Freescale (Motorola)
    - Freescale 68HC08
    - Freescale 68HC11
- Intel
    - Intel 8008
    - Intel 8080 (*8008 source compatible*)
    - Intel 8085 (*8080 binary compatible*)
    - Intel 8051 (*Harvard architecture*)
- RCA
    - RCA 1802
- Zilog
    - Zilog Z80 (*8080 binary compatible*)
    - Zilog Z180 (*Z80 binary compatible*)
    - Zilog Z8
    - Zilog eZ80 (*Z80 binary compatible*)
- Motorola
    - Motorola 6800
    - Motorola 6803
    - Motorola 6809 (*partially 6800 compatible*)
- MOS Technology
    - MOS Technology 6502
- PIC microcontroller
    - Microchip PIC10
    - Microchip PIC12
    - Microchip PIC16
    - Microchip PIC18