Chapter 11
Processor Families

11.1. The main ideas of conditional execution of ARM instructions (see Sections 3.1.2 and B.1) and conditional execution of IA-64 instructions, called predication (see Section 11.7.2), are very similar.

The differences occur in the way that the conditions are set and stored in the processor, and in the way that they are referenced by the conditionally executed instructions.

In ARM processors, the state is stored in four conventional condition code flags N, Z, C, and V (see Section 3.1.1). These flags are optionally set by the results of instruction execution. The particular condition, which may be a function of more than one flag, is named in the condition field of each ARM instruction (see Figure B.1 and Table B.1).

In the IA-64 architecture, there are no conventional condition code flags. Instead, the result (true or false) of executing a Compare<condition> instruction is stored in one of 64 one-bit predicate registers, as described in Section 11.7.2. Each instruction can name one of these bits in its 6-bit predicate field; and the instruction is executed only if the bit is 1 (true).

11.2. Assume that Thumb arithmetic instructions have a 2-operand format, expressed in assembly language as

OP R*dst*,R*src*

as discussed in Section 11.1.1

Also assume that a signed integer Divide instruction (DIV) is available in the Thumb instruction set with the assembly language format

DIV R*dst*,R*src*

This instruction performs the operation [R*dst*]/[R*src*]. It stores the quotient in R*dst* and stores the remainder in R*src*.

Under these assumptions, a possible Thumb program would be:

LDR	R0,G	
LDR	R1,H	
ADD	R0,R1	Leaves $g + h$ in R0.
LDR	R1,E	
LDR	R2,F	
MUL	R1,R2	Leaves $e \times f$ in R1.
DIV	R1,R0	Leaves $(e \times f)/(g + h)$ in R1.
LDR	R0,C	
LDR	R2,D	
DIV	R0,R2	Leaves c/d in R0.
ADD	R0,R1	Leaves denominator in R0.
LDR	R1,A	
LDR	R2,B	
ADD	R1,R2	Leaves $a + b$ in R1.
DIV	R1,R0	Leaves result in R1.
STR	R1,W	Stores result in w.

This program requires 16 instructions as compared to 13 instruction words (some combined instructions) in the HP3000.

11.3. The following table shows some of the important areas for similarity/difference comparisons.

MOTOROLA 680X0	INTEL 80X86
8 Data registers and 8 Address registers (including a processor stack register)	8 General registers (including a processor stack register)
CISC instruction set with flexible addressing modes	CISC instruction set with flexible addressing modes
Large instruction set with multiple-register load/store instructions	Large instruction set with multiple-register push/pop instructions
Memory-mapped I/O only	Separate I/O space as well as memory-mapped I/O
Flat address space	Segmented address space
Big-endian addressing	Little-endian addressing

There is roughly comparable capability and performance between pairs from these two families; that is 68000 vs. 8086, 68020 vs. 80286, 68030 vs. 80386, and 68040 vs. 80486. The cache and pipelining aspects for the high end of each family are summarized in Sections 11.2.2 and 11.3.3.

11.4. An instruction cache is simpler to implement, because its entries do not have to be written back to the main memory. A data cache must have a provision for writing any changed entries back to the memory, before they are overwritten by new entries. ¿From a performance standpoint, a single larger instruction cache would be advantageous only if the frequency of memory data accesses were very low. A unified cache has the potential performance advantage that the proportions of instructions and data vary automatically as a program is executed. However, if separate instruction and data caches are used, they can be accessed in parallel in a pipelined machine; and this is the major performance advantage.

11.5. Memory-mapped I/O requires no specialized support in terms of either instructions or bus signals. A separate I/O space allows simpler I/O interfaces and potentially faster operation. Processors such as those in the IA-32 family, that have a separate I/O space, can also use memory-mapped I/O.

11.6. MOTOROLA - The Autoincrement and Autodecrement modes facilitate stack implementation and accessing successive items in a list. Significant flexibility in accessing structured lists and arrays of addresses and data of different sizes is provided by the displacement, offset, and scale factor features, coupled with indirection.

INTEL - Relocatability in the physical address space is facilitated by the way in which base, index and displacement features are used in generating virtual addresses. As in the Motorola processors, these multiple-component address features enable flexible access to address lists and data structures.

In both families of processors, byte-addressability enables handling of character strings, and the Intel IA-32 String instructions (see Sections 3.21.3 and D.4.1) facilitate movement and processing of byte and doubleword data blocks. The Motorola MOVEM and MOVEP instructions perform similar operations.

11.7. *Flat address space* — Simplest configuration from the standpoint of a single user program and its compilation.

One or more variable-length segments — Efficient allocation of available memory space to variable-length user or operating system programs.

Paged memory — Facilitates automated memory management between the random-access main memory and a sector-organized disk secondary memory (see Chapters 5 and 10). Access privileges can be controlled on a page-by-page basis to ensure protection among users, and between users and the operating system when shared data are involved.

Segmentation and paging — Most flexible arrangement for managing multiple user and system address spaces, including protection mechanisms. The virtual address space can be significantly larger than the physical main memory space.

11.8. ARM program:

Assume that a signed integer Divide instruction is available in the ARM instruction set, and that it has the same format as the Multiply (MUL) instruction (see Figure B.4). The assembly language expression for the Divide (DIV) instruction is

$$\text{DIV} \quad \text{R}d,\text{R}m,\text{R}s$$

and it performs the operation [Rm]/[Rs], loading the quotient into Rm and the remainder into Rd.

```
LDR   R0,C
LDR   R1,D
DIV   R2,R0,R1      Leaves c/d in R0.
LDR   R1,G
LDR   R2,H
ADD   R1,R1,R2      Leaves g + h in R1.
LDR   R2,F
DIV   R3,R2,R1      Leaves f/(g + h) in R2.
LDR   R3,E
MLA   R1,R2,R3,R0   Leaves denominator in R1.
LDR   R0,A
LDR   R2,B
ADD   R0,R0,R2      Leaves a + b in R0.
DIV   R2,R0,R1      Leaves result in R0.
STR   R0,W          Stores result in w.
```

This program requires 15 instructions as compared to 13 instruction words (some combined instructions) in the HP3000.

68000 program (assume 16-bit operands):

MOVE	G,D0	
ADD	H,D0	Leaves $g + h$ in D0.
MOVE	E,D1	
MULS	F,D1	Leaves $e \times f$ in D1.
DIVS	D0,D1	Leaves $(e \times f)/(g + h)$ in D1.
MOVE	C,D0	
EXT.L	D0	See *Note* below.
DIVS	D,D0	Leaves c/d in D0.
ADD	D1,D0	Leaves denominator in D0.
MOVE	A,D1	
ADD	B,D1	
EXT.L	D1	See *Note* below.
DIVS	D0,D1	Leaves result in D1.
MOVE	D1,W	Stores result in w.

Note: The EXT.L instruction sign-extends the 16-bit dividend in the destination register to 32 bits, a requirement of the Divide instruction.

This program contains 14 instructions, as compared to 13 instruction words (some combined instructions) in the HP3000.

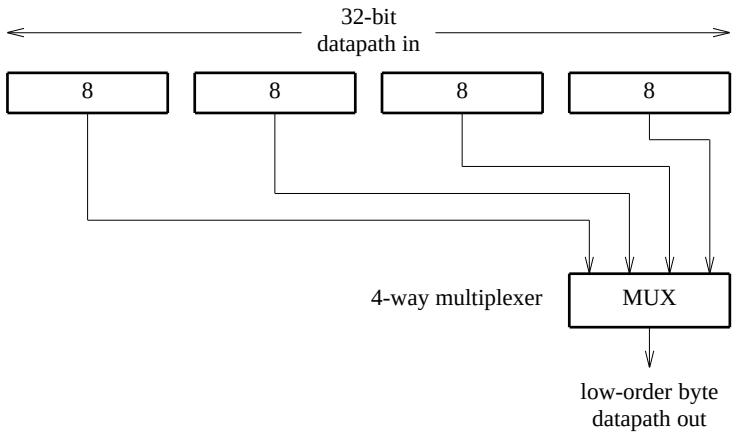
IA-32 program:

MOV	EBX,G	
ADD	EBX,H	Leaves $g + h$ in EBX.
MOV	EAX,E	
IMUL	EAX,F	Leaves $e \times f$ in EAX.
CDQ		See *Note* below.
IDIV	EBX	
MOV	EBX,EAX	Leaves $(e \times f)/(g + h)$ in EBX.
MOVE	EAX,C	
CDQ		See *Note* below.
IDIV	D	Leaves c/d in EAX.
ADD	EBX,EAX	Leaves denominator in EBX.
MOVE	EAX,A	
ADD	EAX,B	Leaves $a + b$ in EAX.
CDQ		See *Note* below.
IDIV	EBX	Leaves result in EAX.
MOV	W,EAX	Stores result in w.

Note: The CDQ instruction sign-extends EAX into EDX (see Section 3.23.1), a requirement of the Divide instruction.

This program contains 16 instructions, as compared to 13 instruction words (some combined instructions) in the HP3000.

11.9. A 4-way multiplexer is required, as shown in the following figure.



11.10. There are no direct counterparts of the memory stack pointer registers SP and FP in the IA-64 architecture. The register remapping hardware in IA-64 processors allows the main program and any sequence of nested subroutines to all use logical register addresses R32 and upward for their own local variables, with the first part of that register space containing parameters passed from the calling routine. An example of this is shown in Figure 11.4.

If the 92 registers of the stacked physical register space are used up by register allocations for a sequence of nested subroutine calls, then some of those physical registers must be spilled into memory to create physical register space for any additional nested subroutines. The memory pointer register used by the processor for that memory area could be considered as a counterpart of SP; but it is not actually used as a TOS pointer by the current routine. In fact, it is not visible to user programs.

11.11. Consider the example of a main program calling a subroutine, as shown in Figure 11.4. The physical register addresses of registers used by the main program are the same as the logical register addresses used in the main program instructions. However, the logical register addresses above 31 used by instructions in the subroutine must have 8 added to them to generate the correct physical register addresses.

The value 8 is the first operand in the Alloc 8,4 instruction executed by the main program. When that instruction is executed, the value 8 is stored in a processor state register associated with the main program. After the subroutine is entered, all logical register addresses above 31 issued by its instructions must be added, in a small adder, to the value (8) in that register. The output of this adder is the physical register address to be used while in the subroutine.

The operand 7 in the Alloc 7,3 instruction executed by the subroutine is stored in a second processor state register associated with the subroutine. The output of that register is added in a second adder to the output of the first adder. After the subroutine calls a second subroutine, logical register addresses above 31 issued by the second subroutine are sent into the first adder. The output of the second adder (logical address + 8 + 7) is the physical register address used while in the second subroutine.

More register/adder pairs are cascaded onto this structure as more subroutines are called. Note that logical register addresses above 31 are always applied to the first adder; and the output of the nth adder is the physical register address to be used in the nth subroutine. All registers and adders are only 7 bits wide because the largest physical register address that needs to be generated is 127.

11.12. Considering cacheing effects only, the average access time over both instruction and data accesses is a function of both cache hit rates and miss penalties (see Sections 5.6.2 and 5.6.3 for general expressions for average access time).

The hit rates in the 21264 L1 caches will be much higher than in the 21164 L1 caches because the 21264 caches are eight times larger. Therefore, the average access time for accesses that can be made on-chip will be larger in the 21164 because of the miss penalty in going to its on-chip L2 cache.

Next, we need to consider the effect on average access time of going to the off-chip caches in each system. The total on-chip cache capacity (112K bytes in the 21164 and 128K bytes in the 21264) is about the same in both the systems. Therefore, we can assume about the same hit rate for on-chip accesses; so the effect on average access time of the miss penalties in going to the off-chip caches will be about the same in each system.

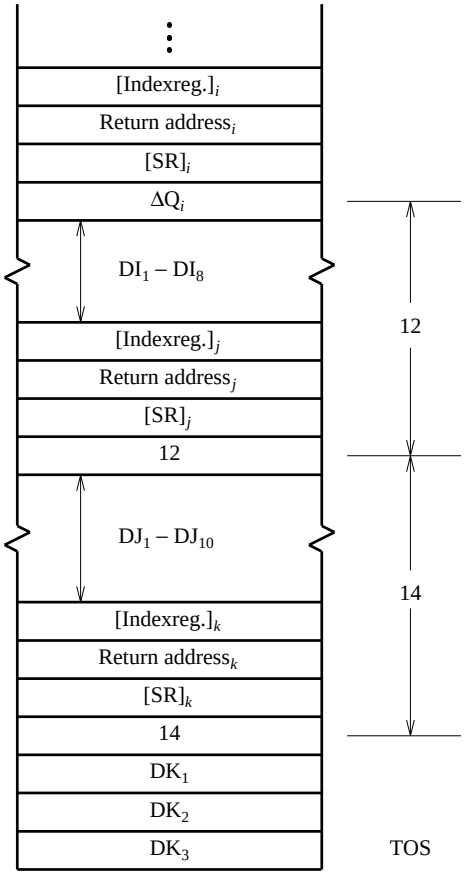
Finally, if the off-chip caches have about the same capacity, the effect on average access times of the miss penalties in going to the main DRAM memories will be about the same in each sytem.

The net result is that average access times in the 21264 should be shorter than in the 21164, leading to faster program execution, primarily because of the different arrangements of the on-chip caches.

11.13. HP3000 program:

```
LOAD    A
LOAD    B
MPYM    C
LOAD    D
MPYM    E
ADD
LOAD    F
MPYM    G
LOAD    H
MPYM    I
DIV
DEL             Combined with previous
                  instruction.
ADD
MPY             Combined with previous
                  instruction.
STOR    W
```

11.14. $\mathrm{Procedure}_i$ generates 8 words of data, $\mathrm{Procedure}_j$ generates 10 words of data, and $\mathrm{Procedure}_k$ generates 3 words of data. Then, the top words in the stack have the following contents:



11.15. HP3000 program:

```
LOAD    A
ADDM    B
LOAD    C
ADDM    D
MPY
LOAD    D
MPYM    E
ADD
STOR    W
```

ARM program:

```
LDR     R0,A
LDR     R1,B
ADD     R0,R0,R1
LDR     R1,C
LDR     R2,D
ADD     R1,R1,R2
LDR     R3,E
MUL     R2,R2,R3
MLA     R0,R0,R1,R2
STR     R0,W
```

68000 program (assume 16-bit operands):

```
MOVE    A,D0
ADD     B,D0
MOVE    C,D1
ADD     D,D1
MULS    D1,D0
MOVE    D,D1
MULS    E,D1
ADD     D1,D0
MOVE    D0,W
```

IA-32 program:

```
MOV     EAX,A
ADD     EAX,B
MOV     EBX,C
ADD     EBX,D
IMUL    EAX,EBX
MOV     EBX,D
IMUL    EBX,E
ADD     EAX,EBX
MOV     W,EAX
```

11.16. Four

11.17. Four and two