The Instruction Set Architecture Level

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

Contents

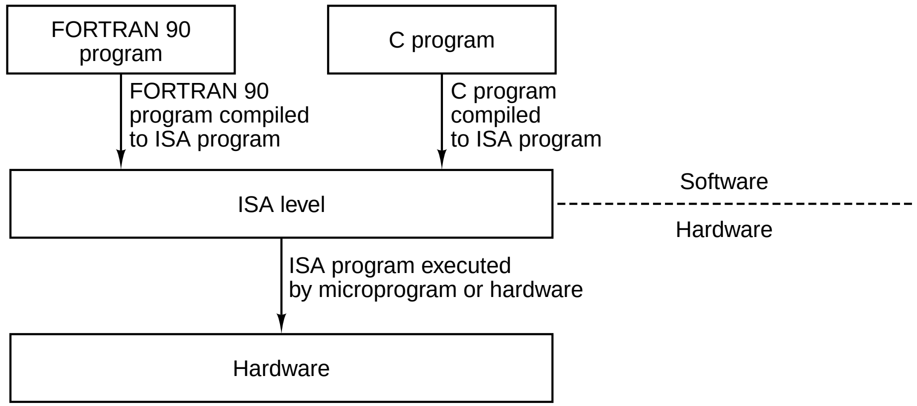
1. Overview.
2. Data Types.
3. Instruction Formats.
4. Addressing.
5. Instruction Types.
6. A Pentium 4 Program.
7. The Intel IA-64.

Overview

The Instruction Set Level

Originally, the only architecture level.

- Also called: "architecture" or "machine language".
  - Target of compilers of high-level languages.
  - Compromise between wishes of hardware engineers and of compiler writers.
- Backward compatibility: ISA of new computer embeds old ISA.
  - Old programs run without change on new computer.



Properties of the ISA Level

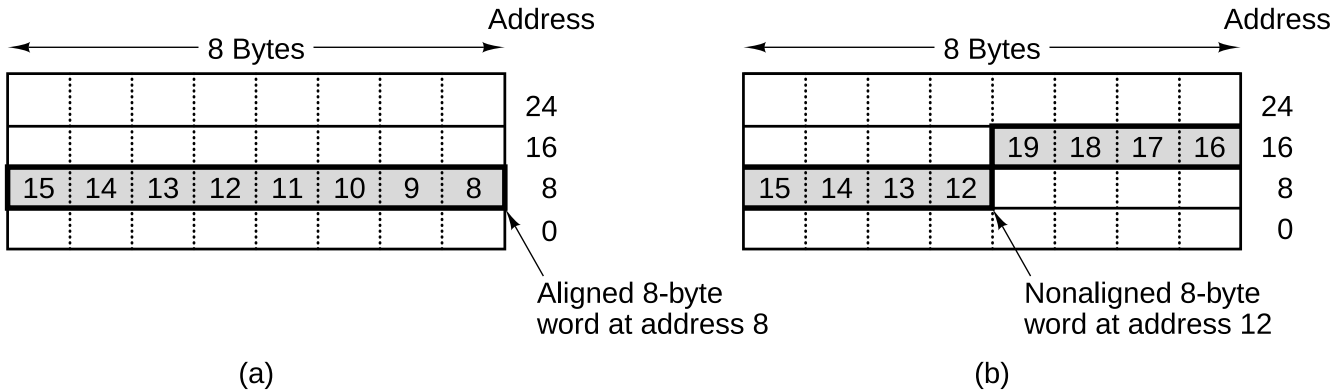
Features that are important for a compiler.

- Various components.
  - Memory model.
  - Registers.
  - Data types and instructions.
- ISA level often formally specified.
  - SPARC V9, JVM.
  - Multiple chip vendors for SPARC processors; multiple JVM implementations.
  - No formal definition of Pentium 4 ISA: only Intel can produce it.
- Often two execution modes.
  - Kernel mode: all instructions are allowed; intended to run operating system.
  - User mode: some instructions are forbidden; intended to run application programs.

Memory Models

All computers divide memory in cells that have consecutive addresses.

- Today: memory cells of 8 bits (bytes).
  - Originally: 7 bit ASCII character plus parity bit.
- Bytes are grouped into 4-byte (32-bit) or 8-byte (64-bit) words.
  - Words are often required to be aligned on natural address boundaries.
  - Memories operate more efficiently if accessed that way.



Registers

Not all microarchitecture registers are visible on ISA level.

- Special-purpose registers: program counter, stack pointer.
- General-purpose registers: rapid access to heavily-used data.
  - Local variables and intermediate calculation results.
  - Compilers and OS adopt convention how registers are used.
    - Some registers hold procedure parameters, others are scratch registers.
- Kernel registers: only available in kernel mode.
  - Used by operating system to control caches, memory, I/O devices.
- PSW (Program Status Word): various bits needed by CPU.
  - Condition codes: set on every ALU cycle to reflect status of most recent operation.
    - Result was wegative (N), result was zero (Z), result caused overflow (V), ...
    - Used by comparison and conditional branch instructions.

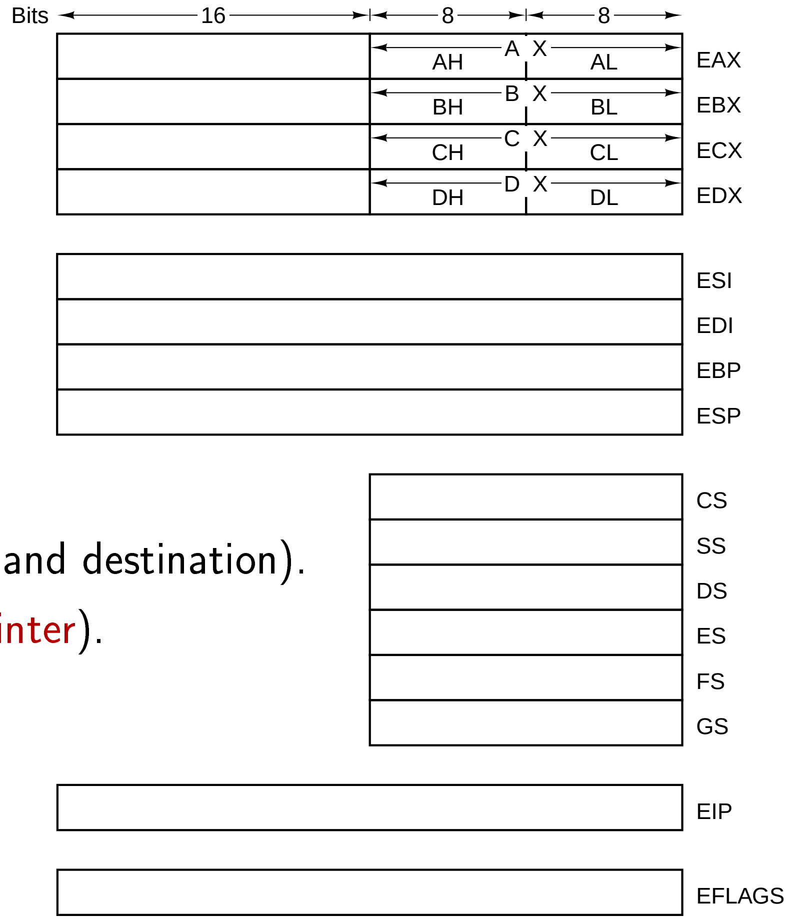
Pentium 4 ISA Level

IA-32 architecture: 32-bit architecture starting with the 80386.

- 3 operating modes.
  - Real mode: Pentium 4 behaves exactly like 8088.
  - Virtual 8086 mode: Pentium 4 runs 8088 code in protected way.
    - Special isolated environment: if program crashes, OS is notified.
    - Used in MS Windows when MS-DOS window is started.
  - Protected mode: normal mode with 4 PSW-controlled privilege levels.
    - Level 0: kernel mode (full access to machine).
    - Level 3: user mode (application programs).
- 2^{32} bytes address space.
  - In each of 16,384 segments (not used by Unix or Windows).
  - Byte-addressed, 32 bit words, little-endian format.

Pentium 4 Registers

- Four general-purpose registers: EAX, EBX, ECX, EDX.
  - EAX is the main arithmetic registers.
  - EDX is needed for multiplication/division.
    - EAX and EDX hold 64-bit products/dividends.
  - Each register holds 16-bit register and 8-bit registers.
    - Compatibility with 8088 and 80286.
- Special-purpose registers.
  - ESI and EDI: string manipulation instructions (source and destination).
  - EBP: points to base of current stack frame (frame pointer).
  - ESP: points to top of stack (stack pointer).
  - EIP: program counter.
  - EFLAGS: program status word.
- Segment registers: CS, SS, DS, ES, FS, GS.
  - 8088 compatibility.



Data Types

Data Types

- Numeric data types.
  - Integer types: 8, 16, 32, 64 bits (counting and identification).
  - Floating-point types: 32, 64, 128 bits (measuring).
  - Often separate registers for integer data and floating-point data.
  - Some computers support decimal numbers (2 decimal digits per byte).
- Nonnumeric data types.
  - Characters: ASCII (7 bits), UNICODE (16 bits).
  - Strings: arrays of characters.
  - Boolean values: bytes 0 and 1.
  - Bit maps: array of boolean values (32-bit word = 32 booleans).
  - Pointers: machine address.

Other data types have to be implemented in software.

Data Types on the Pentium 4

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed Integer	×	×	×		
Unsigned Integer	×	×	×		
Binary Coded Decimal Integer	×				
Floating Point			×	×	

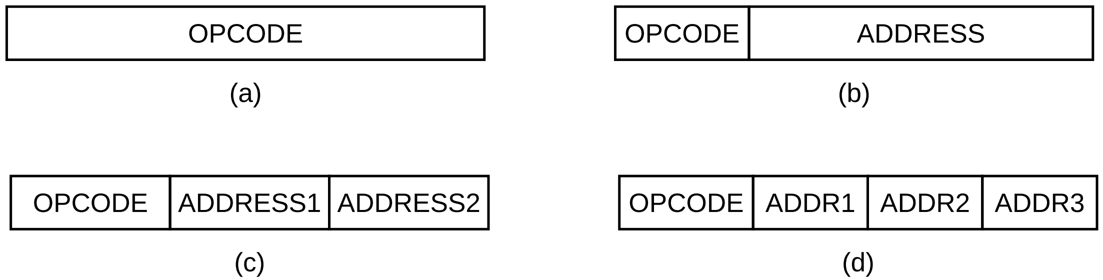
- Arithmetic instructions also on 8 and 16 bit integers.
- Operations do not have to be aligned in memory.
  - Better performance if word addresses are multiples of 4 bytes.
- Operations for copying and searching character strings.
  - Strings whose length are known as well as strings whose end is marked.
  - Used in string manipulation libraries.

Instruction Formats

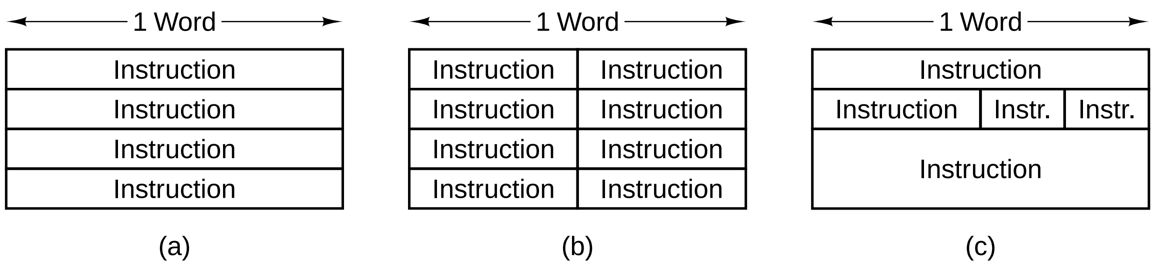
Instruction Formats

Instruction consists of opcode and addresses operands.

- Zero to three addresses.



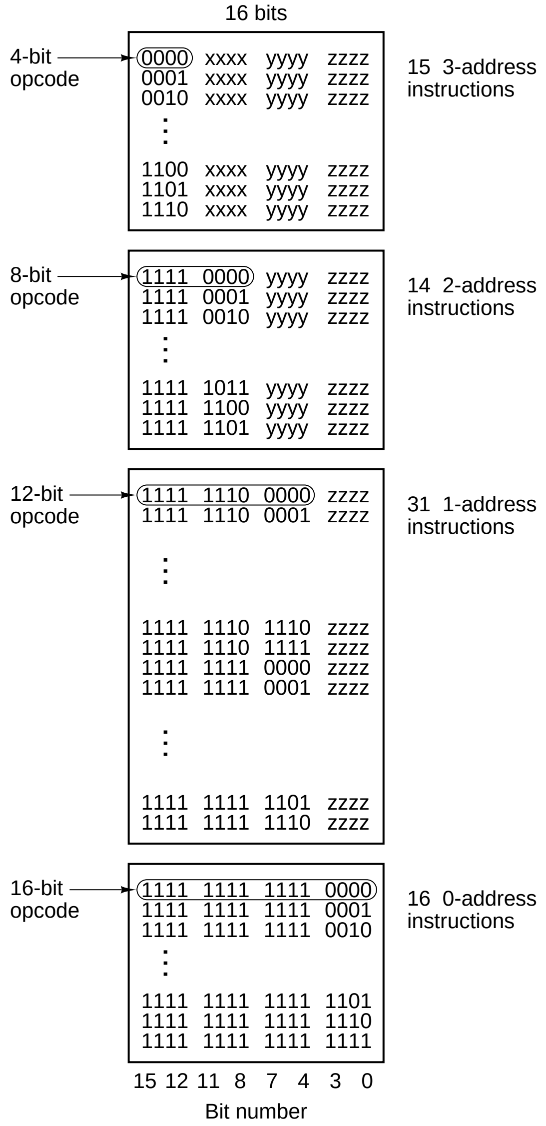
- Instructions may or may not have same length.



Expanding Opcodes

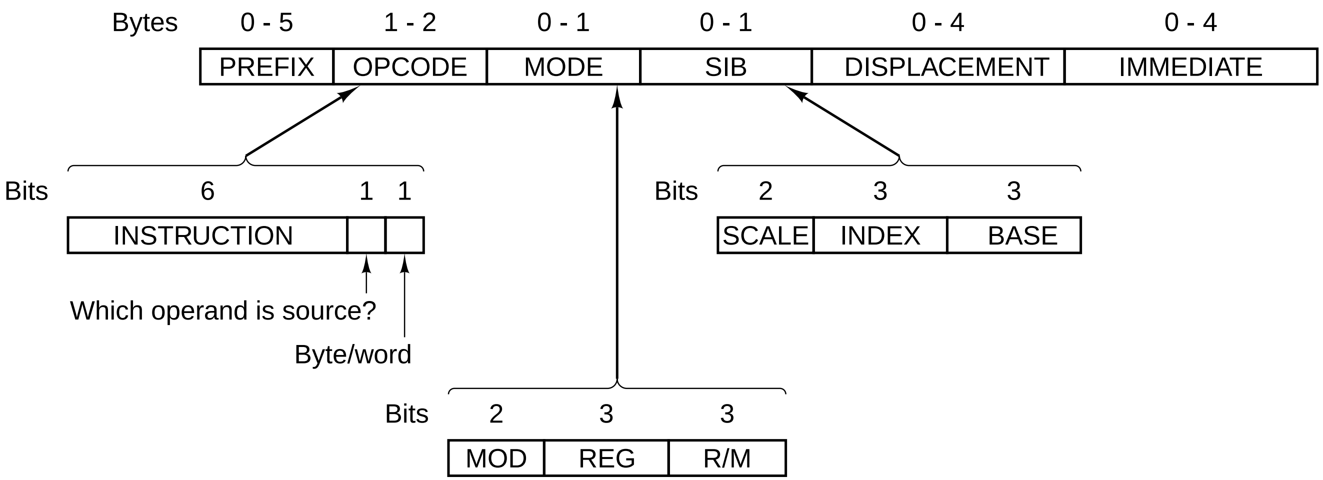
Size of opcode versus size of operand fields.

- 4 bit opcode except 1111.
  - 15 3-address instructions.
- 8 bit opcode 1111 xxxx except 1111 111x.
  - 14 2-address instructions.
- 12 bit opcode 1111 111x xxxx except 1111 1111 1111.
  - 31 1-address instructions.
- 16 bit opcode 1111 1111 1111 xxxx.
  - 16 0-address instructions.



Variable-length opcode to design instruction set.

The Pentium 4 Instruction Format



- Highly complex and irregular with up to six variable-length fields.
  - Reflects long evolution history (and some poor design decisions).
  - Single byte opcode, prefix byte to change action, escape code for second opcode byte.
- For instance: 2 operand instructions.
  - Add two registers, add register to memory, add memory to register.
  - Not: add memory word to another memory word.

Addressing

Addressing

Main part of instruction specifies where operands come from.

- ADD instruction: $a = b + c$ (two sources and one destination).
  - Naive specification: 8-bit opcode and three 32-bit addresses.
- Goal: reduce the size of specification.
  1. Move operands to registers: $r_1 = r_2 + c$.
     - Faster access possible; fewer bits required to specify operands.
     - Explicit LOAD required.
       - Only pays off, if loaded operand is used more than once.
  2. Specify operand implicitly: $r = r + c$.
     - Use operand as a source and a destination.
     - May require to move original value of r to other register.

Various addressing modes possible.

Addressing Modes

How are bits of an address field interpreted to find the operand?

1. Immediate addressing.
2. Direct addressing.
3. Register addressing.
4. Register indirect addressing.
5. Indexed addressing.
6. Based-indexed addressing.
7. Stack addressing.
8. Addressing modes for branch instructions.

Addressing Modes

- Immediate addressing:
  - Address part of operand contains operand itself.
  - `MOV R1, #4`:

MOVI	1	4

  - Load constant 4 to register 1.
  - Only small integer constants can be specified in this way.
- Direct addressing:
  - Give full address of operand in memory.
  - `MOV R1, #A`:

MOVA	1	213474

  - Load word from address of static variable A to register 1.
- Register addressing.
  - Specify register number rather than address.
  - `MOV R1, R2`:

MOVR	1	2

  - Copy content of register 2 to register 1.

Register Indirect Addressing

Operand address is not contained in instruction but in a register.

- Operand address is a pointer.
  - ADD R1, (R2): add to register R1 word at address contained in R2.

ADDRI	1	2

  - Can refer to different addresses in different instruction.
  - Example: assembly code for adding the elements of an array.

```
      MOV R1, #0      ; accumulate sum in R1, initially 0
      MOV R2, #A      ; R2 = address of the array A
      MOV R3, #A+4096 ; R3 = address of first word beyond A
LOOP: ADD R1, (R2)    ; register indirect through R2 to get operand
      ADD R2, #4      ; increment R2 by one word (4 bytes)
      CMP R2, R3      ; are we done yet?
      BLT LOOP        ; if R2 < R3, we are not done, so continue
```

Indexed Addressing

Memory is addressed by giving a register plus a constant offset.

- Example: processing of static arrays.
  - `MOV R4, A(R2)`: load into R4 word whose address has offset A from content of R2.
  - Array is at a fixed address; register contains current index.

MOVIA	4	2	12430

  - Example: assembly code for computing $\sum_i A_i * B_i$.

```
      MOV R1, #0    ; accumulate the sum in R1, initially 0
      MOV R2, #0    ; R2 = index i
      MOV R3, #4096 ; R3 = first index value not in use
LOOP: MOV R4, A(R2) ; R4 = A[i]
      MUL R4, B(R2) ; R4 = A[i] * B[i]
      ADD R1, R4    ; sum all the products into R1
      ADD R2, #4    ; i = i+4 (1 word = 4 bytes)
      CMP R2, R3    ; are we done yet?
      BLT LOOP      ; if R2 < R3, we are not done, so continue
```

Based-Indexed Addressing

Address is computed by sum of two registers plus optional offset.

- Processing of dynamic arrays.
  - `MOV R4, (R2+R5)`: load inot R4 word whose address is the sum of R2 and R5.
  - R5 is the base address of the array.
  - R2 is the current index.
  - Replace loop code in previous example as follows:

MOVBIA	4	2	5

```
       ...
       MOV R5, #A        ; R5 = address of A
       MOV R6, #B        ; R6 = address of B
LOOP:  MOV R4, (R2+R5)  ; R4 = A[i]
       MUL R4, (R2+R6)  ; R4 = A[i] * B[i]
       ...
```

Stack Addressing

Zero-address instructions use stack to avoid explicit memory addresses.

- Example: code for evaluation of $(8 + 2 \times 5)/(1 + 3 \times 2 - 4)$.
  - Reverse Polish notation: $8\ 2\ 5 \times +\ 1\ 3\ 2 \times +4 - /$.

Step	Remaining String	Instruction	Stack
1	$8\ 2\ 5 \times +\ 1\ 3\ 2 \times +\ 4 - /$	BIPUSH 8	8
2	$2\ 5 \times +\ 1\ 3\ 2\ \times +\ 4 - /$	BIPUSH 2	8, 2
3	$5 \times +\ 1\ 3\ 2 \times +\ 4 - /$	BIPUSH 5	8, 2, 5
4	$\times +\ 1\ 3\ 2 \times +\ 4 - /$	IMUL	8, 10
5	$+\ 1\ 3\ 2 \times +\ 4 - /$	IADD	18
6	$13\ 2 \times +\ 4 - /$	BIPUSH 1	18, 1
7	$3\ 2 \times +\ 4 - /$	BIPUSH 3	18, 1, 3
8	$2 \times +\ 4 - /$	BIPUSH 2	18, 1, 3, 2
9	$\times +\ 4 - /$	IMUL	18, 1, 6
10	$+\ 4 - /$	IADD	18, 7
11	$4 - /$	BIPUSH 4	18, 7, 4
12	$-/$	ISUB	18, 3
13	$/$	IDIV	6

Addressing Modes for Branch Instructions

How to specify target address of branch instructions/procedure calls?

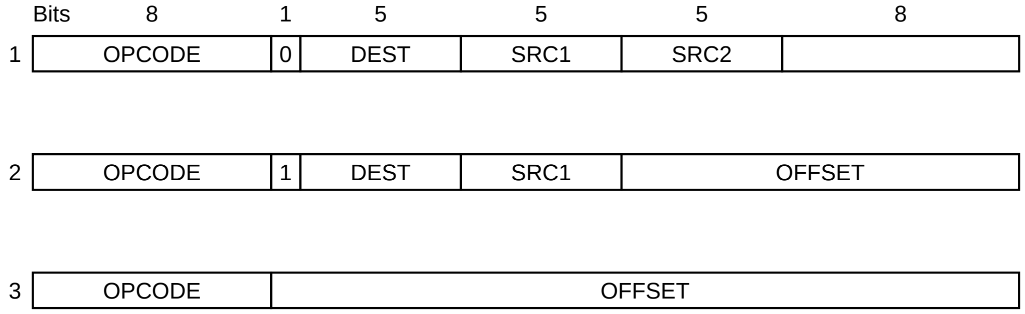
- Direct addressing: unconditional branches (gotos).
  - Generated from conditionals and loops.
- Register indirect addressing or indexed mode.
  - Program may compute target address (computed goto, switch).
- PC-relative addressing: indexed mode where PC acts as register.
  - Target address is specified as offset to current instruction.

Modes presented so far are also useful for branch instructions.

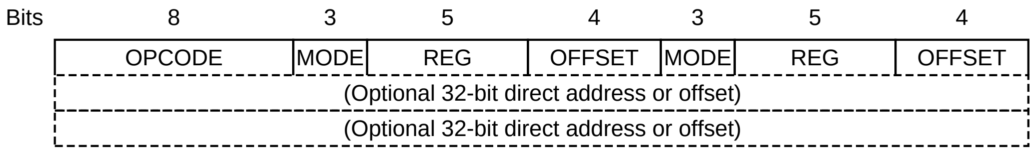
Orthagonality of Opcodes and Addressing Modes

In a clean design, every opcode should permit every addressing mode.

- Three-address machine:
  - Two formats selected by bit.
  - 1 special format for branches.



- Two-address machine:
  - Each operand specified by 12 bits.
  - Mode, register, offset.
  - Optional 32-bit word for address.



In reality, instruction sets are often not that clean.

The Pentium 4 Addressing Modes

Highly irregular structure.

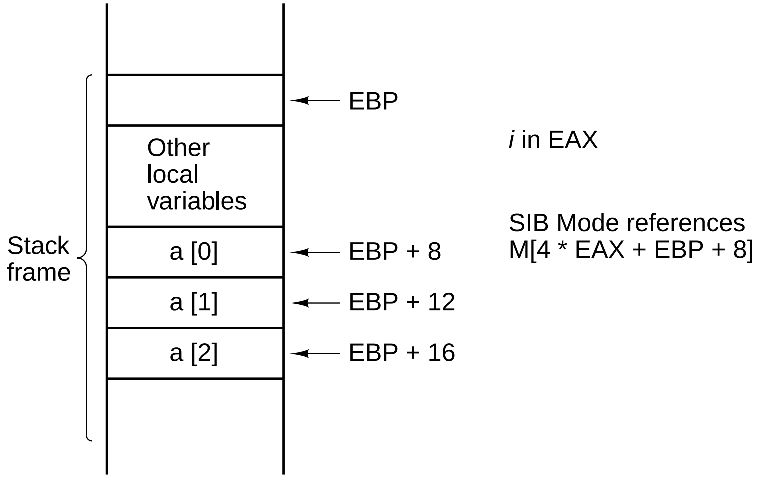
- 32-bit addressing modes.
  - Addressing modes controlled by MODE byte.
  - One operand specified by combination of MOD and R/M.
  - Other operand is register specified by REG.

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX+OFFSET8]	M[EAX+OFFSET32]	EAX or AL
001	M[ECX]	M[ECX+OFFSET8]	M[ECX+OFFSET32]	ECX or CL
010	M[EDX]	M[EDX+OFFSET8]	M[EDX+OFFSET32]	EDX or DL
011	M[EBX]	M[EBX+OFFSET8]	M[EBX+OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP+OFFSET8]	M[EBP+OFFSET32]	EBP or CH
110	M[ESI]	M[ESI+OFFSET8]	M[ESI+OFFSET32]	ESI or DH
111	M[EDI]	M[EDI+OFFSET8]	M[EDI+OFFSET32]	EDI or BH

The Pentium 4 Addressing Mode

In some modes, a SIB byte follows the mode byte.

- SIB (Scale, Index, Base): specifies scale factor and two registers.
  - Operand address is computed by multiplying index register by SCALE (1, 2, 4, 8), adding it to the base register, and (depending on MOD) adding a displacement (8 or 32-bit).
  - Useful for array processing: `for (i = 0; i < n; i++) a[i] = 0;`



Instruction Types

Instruction Types

Which kind of instruction is denoted by the opcode?

1. Data movement instructions.
2. Dyadic operations.
3. Monadic operations.
4. Comparisons and conditional branches.
5. Procedure call instructions.
6. Loop control.
7. Input/output.

Data Movement Instructions

Copy data from one place to another.

- Assignment of values to variables.
  - `A = B;`
  - Copy value at memory address B to location A.
- Prepare data for efficient access and use.
  - Two possibles sources and destinations (memory or register).
  - LOAD to go from memory to register.
  - STORE to go from register to memory.
  - MOVE to go from register to another register.
  - Usually no instruction to copy from memory to memory.

Amount to be moved is usually exactly one word.

Dyadic Operations

Combine two operands to produce a result.

- Arithmetic instructions.
  - Integer and floating-point arithmetic.
- Boolean instructions.
  - AND, OR, NOT; sometimes XOR, NOR, NAND.
  - Important for setting/extracting bits from words.
  - Example: extract second byte from 32 bit word.

10110111 10111100 11011011 10001011	A
00000000 11111111 00000000 00000000	B (mask)
00000000 10111100 00000000 00000000	A AND B
00000000 00000000 00000000 10111110	(A AND B) $>> 16$

Monadic Operations

Take one operand and produce one result.

- Shift or rotate contents of a word.
  - Shift: bits shifted off the end of the word are lost.
  - Rotate: bits shifted off the end of of the word reappear on the other end.

```
00000000 00000000 00000000 01110011 A
00000000 00000000 00000000 00011100 A shifted right 2 bits
11000000 00000000 00000000 00011100 A rotated right 2 bits
```

- Right shift with sign extension.
  - Bits on the left are filled with value of highest bit.

```
11111111 11111111 11111111 11110000 A
00111111 111111111 11111111 11111100 A shifted without sign extension
11111111 111111111 11111111 11111100 A shifted with sign extension.
```

Used to speed up multiplication by powers of 2.

Comparisons and Conditional Branches

Alter the sequence of instructions based on a test result.

- Usually performed by two instructions:
  - Test some condition.
  - If condition is met, branch to a particular memory address.
- Test instruction:
  - Is a bit 0 or not?
  - Is a word 0 or not?
  - Compare two words for equality or size.
- Conditional branch instruction:
  - Previous test instruction sets condition bit.
  - Branch instruction tests the bit and branches, if it is set.

Procedure Call Instructions

Invoke group of instructions to perform a certain task.

- When procedure has finished its task, it must return to the caller.
  - Return address must be stored for the time of the invocation.
- There are various places to store a return address:
  - Fixed memory location: procedure cannot call another procedure.
  - First word of procedure: procedure cannot call itself recursively.
  - Register: leave task to store it in save place to register.
  - Stack: caller pushes return address on stack, procedure pops it from stack.

Return address is usually stored on the stack.

Loop Control

Support to execute a group of instruction a fixed number of times.

- Counter is increased/decreased until upper/lower bound.

```
    for (i = 0; i < n; i++) { statements; }

    i = 0;                        i = 0;
L1: if (i >= n) goto L2;          if (i >= n) goto L2;
    statements;               L1: statements;
    i = i+1;                      i = i+1;
    goto L1;                      if (i < n) goto L1;
L2: ...                       L2: ...
```

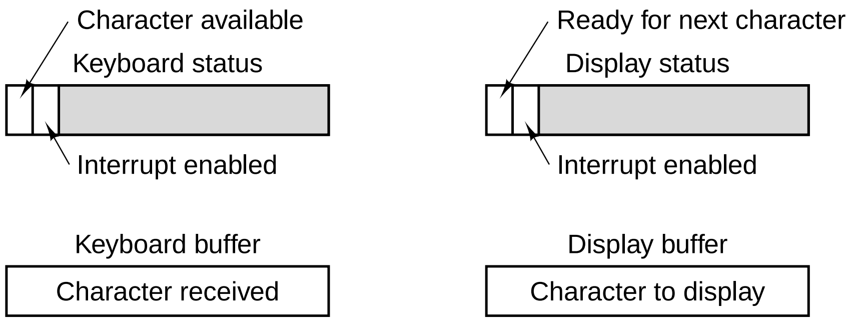
Goal is to minimize number of statements per iterations.

Input/Output

Large variety across different architectures.

- Programmed I/O with busy waiting.
  - Single character is transferred between fixed processor register and selected device.
  - CPU checks in loop whether device has set status bit in processor register.

```
static void output(int buf[], int count) {
  int status, i, ready;
  for (i = 0; i < count; i++) {
    do {
      status = in(DISPLAY_STATUS);
      ready = (status >> 7) & 0x01;
    } while (ready == 0);
    out(DISPLAY_BUFFER, buf[i]);
  }
}
```

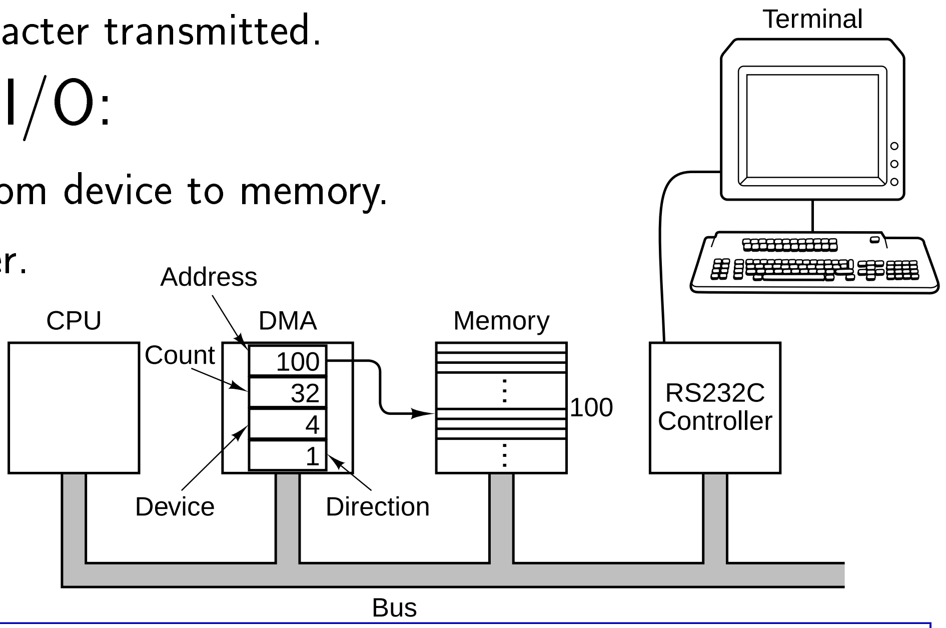


Used only in embedded systems or real-time systems.

Input/Output Instructions

General-purpose computers use interrupt-driven I/O or DMA I/O.

- Interrupt-driven I/O:
  - Device generates interrupt when I/O operation is completed.
  - CPU can execute other programs in the mean time (multi-tasking).
  - Interrupt is generated for each single character transmitted.
- DMA (Direct Memory Access) I/O:
  - DMA controller transfers block of data from device to memory.
  - CPU initializes registers in DMA controller.
  - DMA controller generates interrupt when I/O operation has been finished.



The Pentium 4 Instructions

Very complex instruction set.

- Mixture of instruction sets.
  - 8088 instructions.
  - 32-bit instructions.
- Special support:
  - BCD (binary coded decimal arithmetic).
    - ∗ 8 bit contain two decimal digits.
  - String processsing.

Backward compatibility.

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOV DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination
# = shift/rotate count
LV = # locals

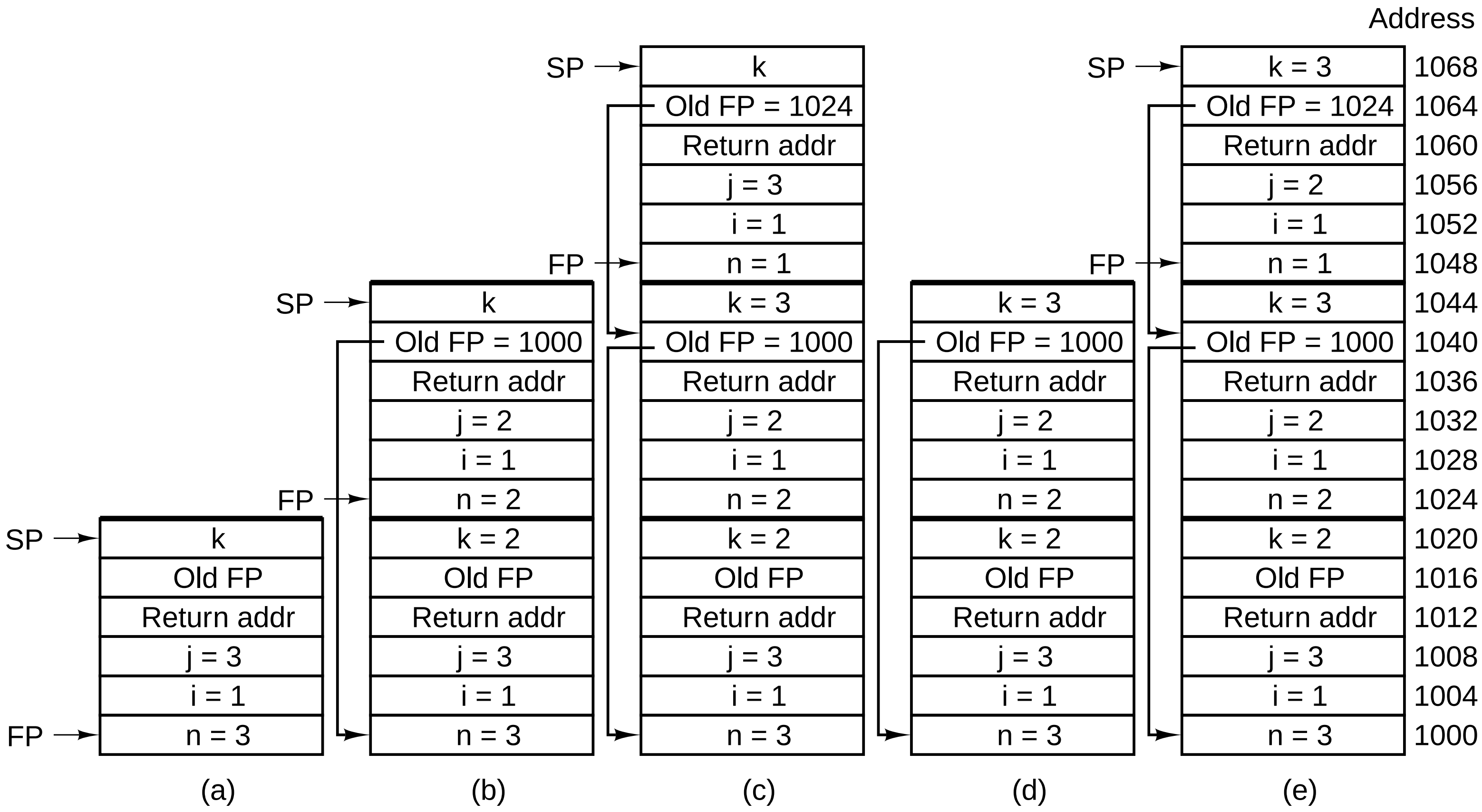
A Pentium 4 Program

Program Example

Towers of Hanoi

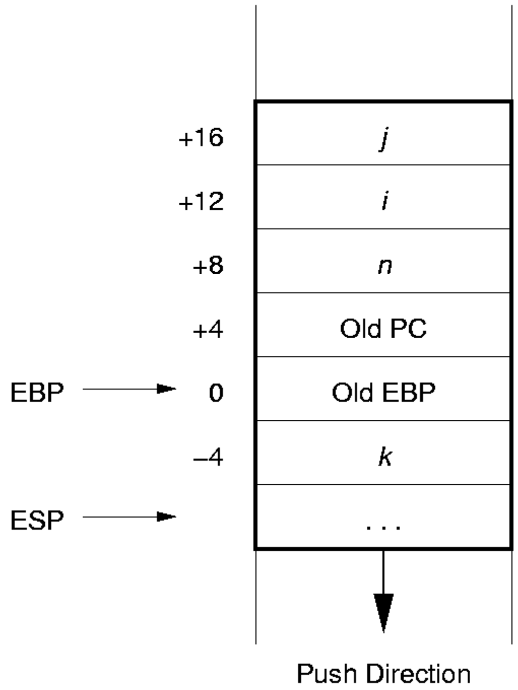
```
static void towers(int n, int i, int j) {
  int k;
  if (n == 1)
    printf("Move disk from %d to %d\n", i, j);
  else {
    k = 6-i-j;
    towers(n-1, i, k);
    towers(1, i, j);
    towers(n-1, k, j);
  }
}
```

Stack View


Address
SP
k
Old FP
Return addr
j = 3
i = 1
n = 3
FP
(a)
SP
k
Old FP = 1000
Return addr
j = 2
i = 1
n = 2
FP
k = 2
Old FP
Return addr
j = 3
i = 1
n = 3
(b)
SP
k
Old FP = 1024
Return addr
j = 3
i = 1
n = 1
FP
k = 3
Old FP = 1000
Return addr
j = 2
i = 1
n = 2
k = 2
Old FP
Return addr
j = 3
i = 1
n = 3
(c)
k = 3
Old FP = 1000
Return addr
j = 2
i = 1
n = 2
k = 2
Old FP
Return addr
j = 3
i = 1
n = 3
(d)
SP
k = 3
Old FP = 1024
Return addr
j = 2
i = 1
n = 1
FP
k = 3
Old FP = 1000
Return addr
j = 2
i = 1
n = 2
k = 2
Old FP
Return addr
j = 3
i = 1
n = 3
(e)
1068
1064
1060
1056
1052
1048
1044
1040
1036
1032
1028
1024
1020
1016
1012
1008
1004
1000

Stack View for the Pentium 4

Previous picture is technically not correct.



Stack actually grows downward, parameter order is reverted.

Stack View for the Pentium 4

- EBP register is used as the frame pointer.
  - First two words are used for linkage (old PC and old EBP).
  - Parameters n, i, j are at EBP+8, EBP+12, EBP+16.
  - Local variable k is at EBP-4.
- Procedure start: new frame is established at end of old one.
  - Stack grows downwards (push: ESP is decreased)
  - Stack pointer ESP is copied to frame pointer EBP.
- Procedure call: parameters are pushed in reverse order.
  - C calling convention.
  - First parameter has constant offset.
  - Number of parameters may be variable
- Procedure return: parameters are popped off the stack.
  - Stack pointer ESP is adjusted (increased).

Pentium 4 Assembly Language Program

```
        .586                                      ; compile for Pentium (not 8088)
.MODEL FLAT
PUBLIC _towers                                    ; export 'towers'
EXTERN _printf: NEAR                              ; import printf
.CODE
_towers: PUSH EBP                                 ; save EBP (frame pointer)
         MOV EBP, ESP                             ; set new frame pointer above ESP
         PUSH 0                                   ; k := 0
         CMP [EBP+8], 1                           ; if (n == 1)
         JNE L1                                   ; branch if n is not 1
         MOV EAX, [EBP+16]                        ; EAX := j
         PUSH EAX                                 ; push j on stack
         MOV EAX, [EBP+12]                        ; EAX := i
         PUSH EAX                                 ; push i on stack
         PUSH OFFSET FLAT:format                  ; push address of format
         CALL _printf                             ; call printf
         ADD ESP, 12                              ; remove params from the stack
         JMP Done                                 ; we are finished
...
```

Pentium 4 Assembly Language Program

```
...
L1:     MOV EAX, 6                  ; EAX = 6
        SUB EAX, [EBP+12]           ; EAX = 6-i
        SUB EAX, [EBP+16]           ; EAX = 6-i-j
        MOV [EBP-4], EAX            ; k = EAX
        PUSH EAX                    ; push k on stack
        MOV EAX, [EBP+12]           ; EAX = i
        PUSH EAX                    ; push i on stack
        MOV EAX, [EBP+8]            ; EAX = n
        DEC EAX                     ; EAX = n-1
        PUSH EAX                    ; push n-1 on stack
        CALL _towers                ; call towers(n-1, i, 6-i-j)
        ADD ESP, 12                 ; remove params from the stack
...
```

Pentium 4 Assembly Language Program

```
...
        PUSH EAX                            ; start towers(n-1, 1, k)
        ...
        ADD ESP, 12                         ; remove params from the stack
        PUSH EAX                            ; start towers(n-1, 6-i-j, i)
        ...
        ADD ESP, 12                         ; remove params from the stack
Done:   LEAVE                               ; prepare to exit
        RET 0                               ; return to the caller
.DATA
format  DB "Move disk from %d to %d\n" ; format string
END
```

The Intel IA-64

The Intel IA-64

The IA-32 line has reached its limits.

- IA-32: wrong properties for current technology.
  - Irregular instructions which are hard to decode.
  - Two-address memory-oriented (rather than register-oriented) ISA.
  - Small and irregular register set.
  - 32 bit addresses limit programs to 4 GB of memory.
    - EMT-64: Pentium 4 with 64-bit registers and address space (Intel Xeon, AMT Opteron).
- IA-64: New 64 bit architecture.
  - Dual mode: also capable of running IA-32 programs.
  - Going to be implemented by a series of CPUs.
  - Currently especially for high-end servers and supercomputers.

The Intel architecture for the "Itanium" processor line.

The IA-64 Model

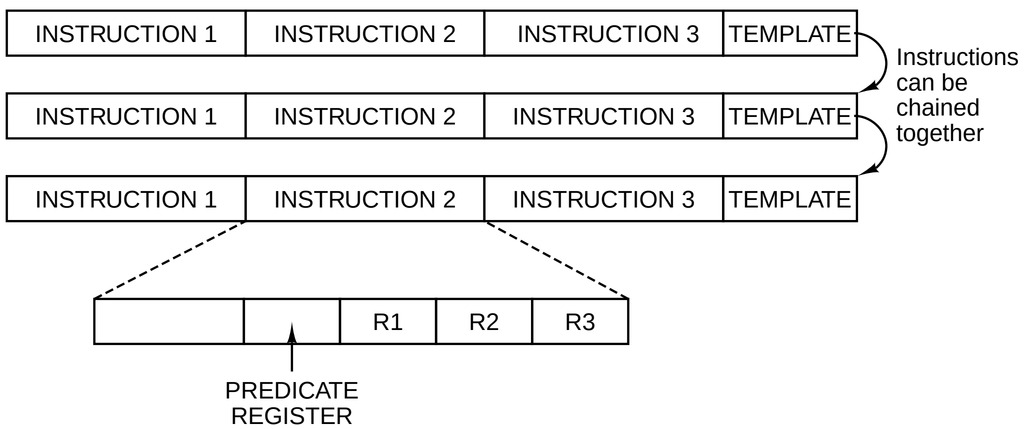
What is new compared to the IA-32?

- Load/store architecture.
  - Instructions operate on registers rather than on memory.
- 64-bit addresses and 64-bit registers.
  - 64 general registers available to all IA-64 programs.
  - Additional registers available to IA-32 programs.
- All instructions have same fixed format.
  - Opcode, two 6-bit source register fields, 6-bit destination register field, 6-bit predicate register.
  - Most instructions take two register operands and put result to destination register.
  - Many functional units for doing different operations in parallel.

Modern architecture in the line of current RISC machines.

EPIC (Explicitly Parallel Instruction Computing)

- Instructions are grouped to bundles.
  - 128-bit bundle contains three 40-bit instructions and 8-bit template.
  - Bundles are chained together by bit in template.
    - Bundles can contain more than three instructions.
  - Template contains scheduling information.
    - Tells CPU which instructions can be executed in parallel.



CPU parallelism is exposed for scheduling at compile-time.

Predication

Reduce the number of conditional branches.

- Predicated instructions:
  - Instruction contains number of predicate register.
  - Instruction is only executed, if predicate register contains 1.
  - Test instruction sets pair of predicate registers to condition and its negation.

```
if (R1 == R2)              CMP R1, R2       CMPEQ R1, R2, P4
  R3 = R4+R5;              BNE L1           <P4> ADD R3, R4, R5
else                       MOV R3, R4       <P5> SUB R6, R4, R5
  R6 = R4-R5;              ADD R3, R5
                           BR L2
                       L1: MOV R6, R4
                           SUB R6, R5
                       L2: ...
```

Processor pipeline can be efficiently utilized.

Speculative Loads

Support for speculative execution.

- Speculative LOAD:
  - LOAD instruction whose result may not be needed.
  - Must not cause exception:
    - Cache miss stops CPU until cache line is loaded.
- Speculative LOAD may fail.
  - If result is not in cache, poison bit is set for loaded register.
- CHECK instruction.
  - Must be inserted by compiler, before speculatively loaded register is used.
  - If poison bit is set, pending exception occurs at that point.

Operands may be fetched in advance without penalty.