Intel Pentium processors

- IA-32 processors
  - from 8086 to Pentium 4
- IA-32 Instruction Set Architecture
  - registers
  - addressing
  - assembly language instructions
  - x87 floating-point unit
  - MMX, SSE and SSE2
- Pentium 4 microarchitecture
  - the NetBurst microarchitecture
  - hyper-threading microarchitecture

IA-32 processors

- **8086**, 1978
  - 8 MHz, no cache
  - 16 bit architecture: 16 bit registers and data bus
  - 20 bit addresses, 1 MB segmented address space
- **Intel 286**, 1982
  - 12.5 MHz, no cache
  - 16-bit registers, 24 bit addesses, 16 MB address space
  - protected mode operation, support for segmented virtual memory
- **Intel 386**, 1985
  - first 32-bit processor in the IA-32 family
  - 20 MHz, no cache, 32-entry 4-way set associative TLB
  - 32-bit addresses, 4 GB address space
  - supports both a segmented and a flat memory model
  - supports virtual memory through paging

Intel 386

- 386 was the first processor based on a micro-architecture
  - instruction execution is separated from the ISA
- Bus interface unit
  - accesses memory
- Code prefetch unit
  - receives code from the bus unit into a 16-byte queue
- Instruction decode unit
  - fetches instructions from prefetch buffer, decodes into microcode
- Execution unit
  - executes microcode instructions
- Segment unit
  - translates logical adresses to linear addresses, protection checking
- Paging unit
  - translates linear addresses to physical addresses, protection, TLA

IA-32 processors (cont.)

- **Intel 486**, 1989
  - 25 MHz, 8 KB L1 cache (write-through)
  - 5-stage pipelined instruction execution
  - integrated x87 FPU
  - support for second level cache
- **Intel Pentium**, 1993
  - 60 MHz, 8+8 KB L1 cache (write-back), 64-bit external bus
  - superscalar design with 2 pipelines
  - branch prediction with on-chip branch table
- Later Pentium processor introduced the MMX technology
  - parallel operations on packed integers in 64-bit MMX registers
  - added 47 new instructions to the instruction set

IA-32 processors (cont.)

- **Pentium Pro**, 1995
  - introduces the P6 microarchitecture
  - 200 MHz, 8+8 KB L1, 256 KB L2 cache
  - dedicated 64-bit backside cache bus connecting CPU with cache
  - 3-way superscalar design
  - IA-32 instructios are decoded into micro-ops
  - out-of-order execution with 5 parallel execution units
  - retirement unit retires completed micro-ops in program order
  - improved branch prediction
  - improved cache performance

IA-32 processors (cont.)

- **Pentium II**, 1997
  - 266 MHz, 16+16 KB L1, 256 KB L2 cache
  - supports 256, 512 KB or 1 MB L2 cache
  - adds MMX technology to the P6 micro-architecture
  - Xeon and Celeron improved cache organization
- **Pentium III**, 1999
  - 500 MHz, 16+16 KB L1, 256 or 512 KB L2 cache
  - introduces Streaming SIMD Extension (SSE) technology
  - parallel operations on packed 32-bit floating-point values in 128-bit SSE registers
  - Pentium III Xeon improved cache performance

IA-32 processors (cont.)

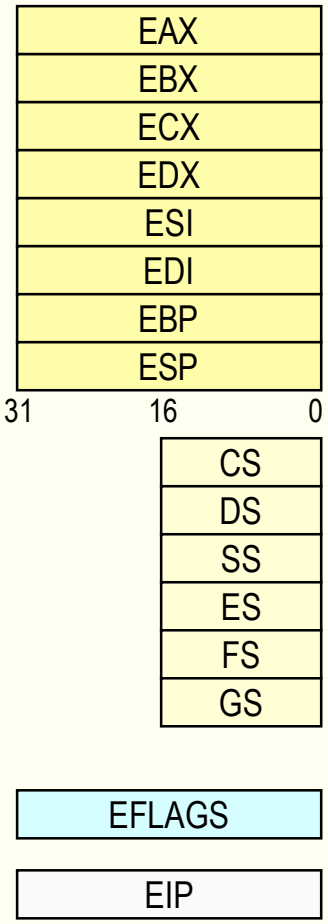
- **Pentium 4**, 2000
  - based on the NetBurst microarchitecture
  - 1.5 GHz, 8 KB L1 data cache, Execution Trace Cache, 256 KB L2 cache
  - 400 MHz pipelined system bus
  - SSE2 extension, parallel operations also on packed 64-bit floating-point values
- **Xeon, 2001**
  - based on the NetBurst microarchitecture
  - introduced Hyper-Threading technology 2002
- **Pentium M**
  - for mobile systems, advanced power management
  - 32+32 KB L1 cache, write-back, 1 MB L2 cache
  - 400 MHz system bus

IA-32 Instruction Set Architecture

- Desribes the basic execution environment of all IA-32 processors
  - describes the facilities for processing instructions and storing data, as seen by the assembly language programmer
  - compatible with older 16-bit architectures
- IA-32 processors support three operating modes
  - protected mode
    - the native state of the processor
    - all instructions and architectural features are available
  - real-address mode
    - implements the programming environment of the 8086 processor
  - system management mode
    - for use in operating systems
    - saves current context, switches to a separate address space

Registers

- 8 general-purpose registers, 32-bit
  - can be used in instruction execution to store operands and addresses
  - ESP is used for stack pointer
- 6 segment registers, 16-bit
  - used to hold segment selectors
- Status flags, 32-bit
  - consists of status bits describing the current status of the processor
- Instruction pointer, 32-bit
  - points to the next instruction to be executed
- 8 floating-point / MMX registers, 64-bit
- 8 XMM registers for SSE operations, 128-bit

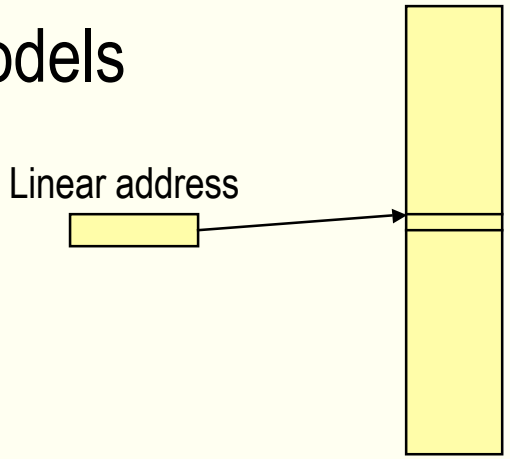


Use of general purpose registers

- Can refer to the lower 16-bit part of the registers with names without the prefix E (AX, BX, CX, ...)
  - can refer to the two lower bytes in the registers as AH, AL, ...
- Registers are used for special purouses in different instructions
  - EAX – Ackumulator for operands and results
  - EBX – pointer to data in DS segment
  - ECX – counter for string and loop operations
  - EDX – I/O pointer
  - ESI – source pointer for string operations, pointer to DS segment
  - EDI – destination pointer for string operations, pointer to ES segment
  - ESP - stack pointer (in the SS segment)
  - EBP - pointer to data on the stack

Memory organization

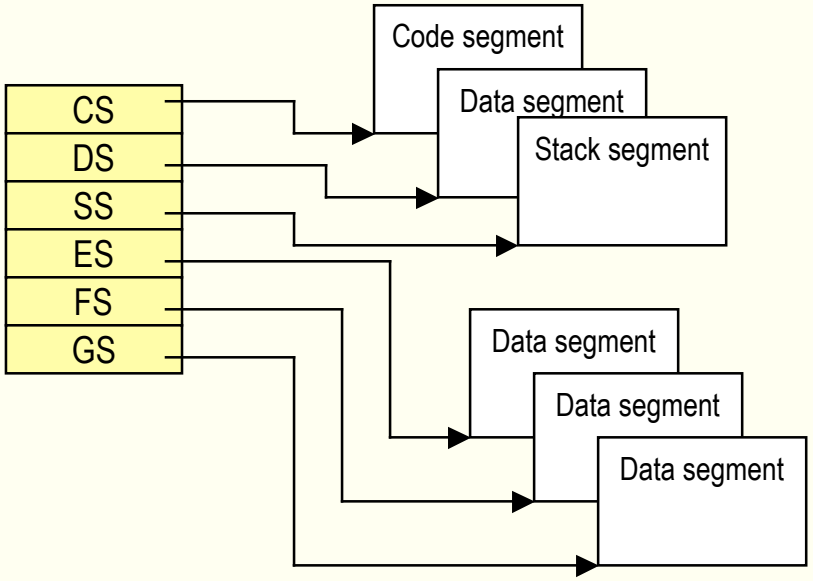
- The IA-32 ISA supports three memory models
- Flat memory
  - linear address space
  - single byte addressable memory
  - contiguous addresses from 0 to 2^{32-1}
  - can be used together with paging



- Segmented memory
  - memory appears as a group of separate address spaces
  - code, data and stack segments
  - uses logical addresses consisting of a segment selector and an offset
  - can be used together with paging
- Real-address mode
  - compatible with older IA-32 processors

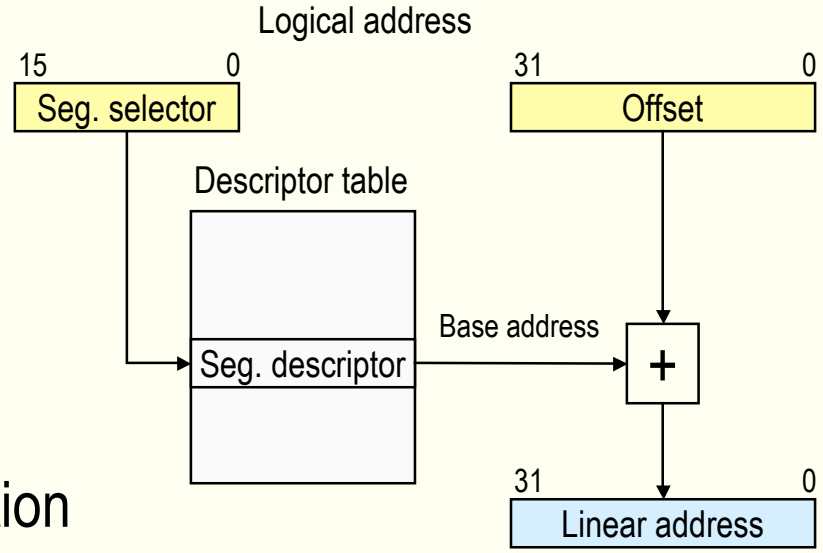
Segment registers

- Segment registers hold 16-bit segment selectors
  - pointer that identifies a segment in memory
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES, FS, GS – used for additional data segments
- Segment selectors point to segment descriptors
  - data structure that describes a segment



Address translation

- Logical adresses consist of
  - a 16-bit segment selector
  - a 32-bit offset
- Two-level address translation
  - logical to linear address translation through segmentation
  - linear to physical address translation through paging

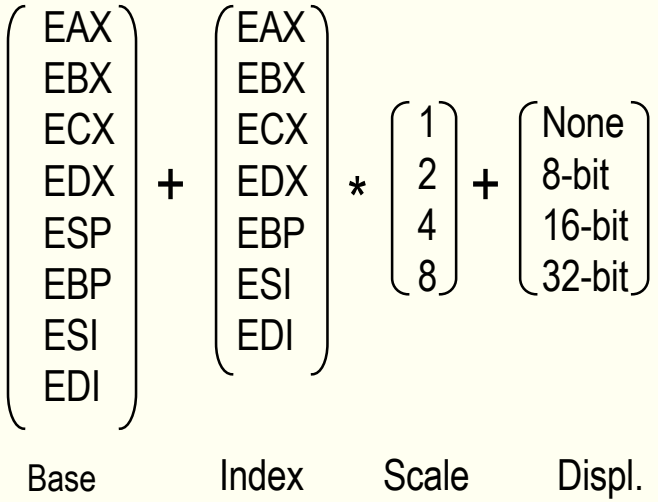


Operand addressing

- IA-32 instructions operate on zero, one or two operands
  - in general, one operand may be a memory reference
- Operands can be
  - immediate
  - a register
  - a memory location
- Some operations (DIV and MUL) use quadword operands
  - represented by register pairs, separated by a colon (EDX:EAX)
- Memory locations are specified by a segment selector and an offset (a far pointer)
  - segment selector are often implicit
    (CS for instruction access, SS for stack push/pop, DS for data references, ES for destination strings)
  - can also be specified explicitely: `mov ES:[EBX], EAX`

Addressing modes

- The offset part of an operand address can be specified as
  - a static value (a displacement)
  - as an address computation of the form
    offset = *Base* + (*Index**2^{Scale}) + *Displacement*
    where
    - *Base* is one of the registers
    - *Index* is one of the registers
    - *Scale* is a constant value
      1, 2 , 4 or 8
    - *Displacement* is a 8, 16 or
      32-bit value



Addressing modes (cont)

- Displacement
  - absolute address
- Base
  - register indirect addressing
- Base + Displacement
  - index into an array , fields of records
- Base + Index + Displacement
  - access two-dimensional arrays
- (Index*scale) + Displacement
  - index arrays with element sizes greater than 1
- Base + (Index*scale) + Displacement
  - access two-dimensional arrays with an element size greater than 1

Instruction set

- Very large instruction set, over 300 instructions
  - CISC-like instruction set
- Instructions can be divided into the following groups
  - data transfer instructions (MOV)
  - binary arithmetic (ADD, SUB)
  - logical instructions (AND, OR)
  - shift and rotate (ROR, SAR)
  - bit and byte instructions (BTS, SETE)
  - control transfer instructions (JMP, CALL, RET)
  - string instructions (MOVS, SCAS)
  - flag control instructions (STD, STC)
  - segment register instructions (LDS)
  - miscellaneous instructions (LEA, NOP, CPUID)

Data transfer instructions

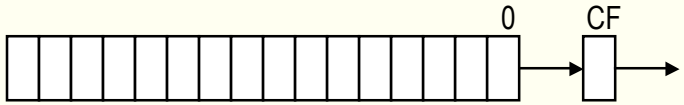
- Move data memory–register or register–register
  - MOV – unconditional move
- Conditional move MOV*cc*, move if a condition *cc* is true
  - CMOVE – conditional move if equal
  - CMOVLE – conditional move if less or equal
- Exchange
  - XCHG – exchange register and memory (or register – register) (atomic instruction, used to implement semaphors)
- Stack operations
  - PUSH, POP
  - PUSHA, POPA – push/pop all general purpose registers
- Conversion
  - CBW – convert byte to word (by sign extension)

Binary arithmetic

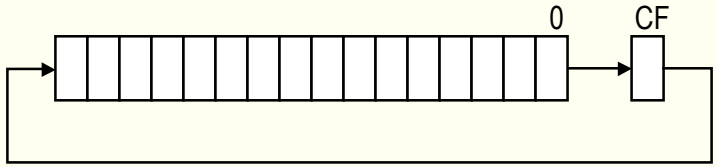
- Addition, subtraction
  - ADD, SUB
  - also add with carry (ADC) and subtract with borrow (SBB)
- Multiplication, division
  - MUL
    - EDX:EAX ← EAX * operand
  - DIV
    - EDX:EAX ← EDX:EAX / operand (EAX quotient, EDX remainder)
  - IMUL, IDIV
    - signed multiply and divide
- Compare
  - CMP – set status flags for use in conditional jump

Logical, shift and rotate instructions

- Bitwise logical AND, OR, XOR
- Negation NOT
- Shift arithmetic right and left SAR, SAL
  - shifts the destination the specified number of bits left/right
  - bits are first shifted into the carry flag and then discarded
  - count is an immediate value or the CL register, masked to 5 bits



- Rotate right and left, ROL, ROR
  - similar as shift, but rotates the bits throug the carry flag
  - no bits are lost



Bit and byte instructions

- Bit test, BT
- Bit Test and Set, BTS
- Bit Scan Forward, BSF
- Bit Scan Reverse, BSR
  - scans the operand for a set bit, stores the index in destination
- SET*cc* – sets a byte to 0 or 1 depending on condition *cc*
  - Set Byte If Equal, SETE
  - Set Byte If Greater Or Equal, SETGE
- TEST – Logical compare
  - does a logical AND of operands and sets status flags
  - does not alter the operands

Control transfer instructions

- Unconditional control transfer
  - JMP, CALL, RET
  - CALL saves the current EIP on the stack, popped by RET
- Conditional control transfer
  - J*cc* – jump if condition *cc* is true
  - JNE – Jump If Not Equal
  - JGE – Jump If Greater Or Equal
- Loop instructions
  - LOOP – conditional jump using ECX as a count

String instructions

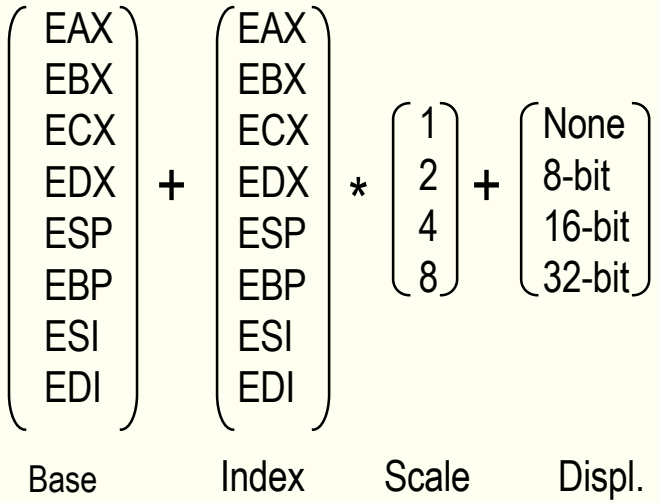
- Operates on contiguos data structures in memory
  - bytes, words or doublewords
- MOVS – Move String
  - ESI contains source address
  - EDI contains destination address
- CMPS – Compare String
- Can be used repeatedly with a count in ECX register

Flag control instructions

- Instructions to modify some of the flags in the EFLAGS status register
- STC – Set Carry Flag
- CLC – Clear Carry Flag
- STD, CLD – Set Direction Flag, Clear Direction Flag
  - controls direction in string operations, etc.

Miscellaneous instructions

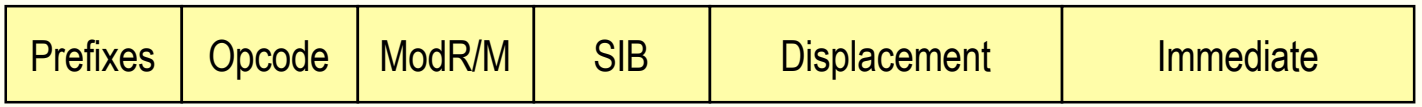
- NOP – No-Operation
- LEA – Load Effective Address
  - computes the effective address of a source operand
  - can be used for exaluating expressions in the form of an address computation



- CPUID – Processor Identification
  - returns information about the type of processor
  - can be used to find out the capabilities of the processor

Instruction format

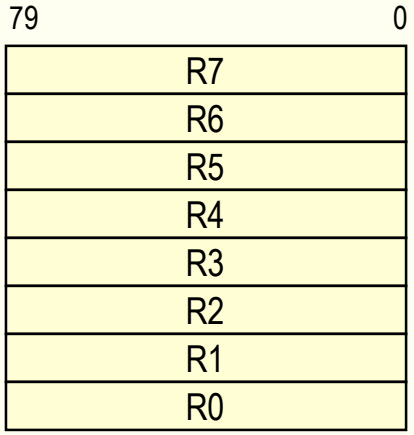
- IA-32 instructions are decoded into opcodes of the following format



  - up to four prefix bytes
    - prefixes for lock/repeat, segment override / branch hint, operand size override, address size override
  - 1–2 opcode bytes
  - 1 byte ModR/M and SIB (optional)
    - describes the addressing mode and register number
  - 1, 2 or 4 bytes displacement
  - 1, 2 or 4 bytes immediate

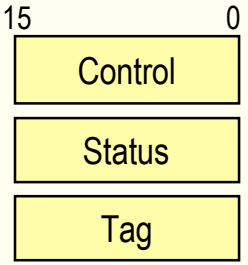
x87 Floating-Point Unit

- Conforms to the IEEE 754 standard
- The floating-point unit is independent of the basic execution environment and of the SSE (and SSE2) execution environment
  - shares state with the MMX execution environment
  - MMX registers are aliased to the floating-point registers
- 8 floating-point data registers, 80-bit
  - 1 sign bit
  - 15 bits exponent
  - 63 bits significand
- Floating-point values are stored in double extended precision (80 bits)
  - automatically converted to double extended when loaded into a register



FPU special-purpose registers

- The FPU has three 16-bit special purpouse registers
  - control register
  - status register
  - tag register



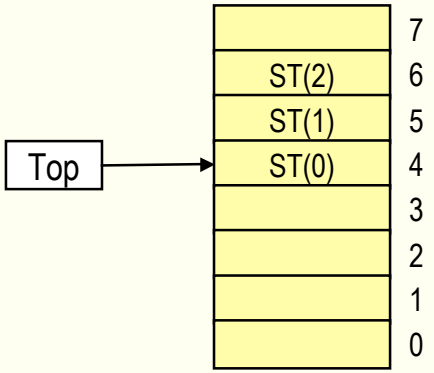
- Control register contains
  - precision control field
    - single, double or double extended precision
    - default is 64 bits precision for mantissa
  - exception mask bits
    - when set, the FPU does not generate exceptions on underflow, overflow, denormal value, divide by zero, ...
  - rounding control field
    - selects one of the four rounding modes

FPU special-purpose registers (cont.)

- Status register contains
  - condition code flags, indicating the result of FP compare and arithmetic operations
  - exception flags, indicating an exeption
  - top-of-stack pointer (3 bits)
- Tag register contains two bits for each register, describing the contents of each register
  - valid number
  - zero
  - special (NaN, infinity, denormal)
  - empty
- Two 48-bit pointers
  - last instruction pointer and last data pointer
- Opcode of last FP instruction

Floating-point data registers

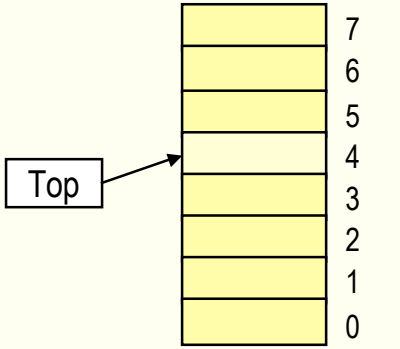
- The eight FPU data registers are treated as a stack
  - references to FP registers are relative to the top of the stack
- The register number of the current top-of-stack is stored in the TOP field in the status word register (3 bits)
  - load operations decrement TOP with 1, modulo 8
  - store operations increment TOP with 1, modulo 8
- Register references are relative to the top-of-stack
  - ST(0) is the top-of-stack
  - ST(1) is top-of-stack + 1
- Most FP instructions implicitely operate on the top-of-stack
  - two-operand instructions use ST(0) and ST(1)
  - one-operand instructions use ST(0)



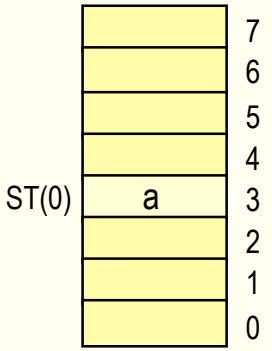
Example: Inner product

```
/* Compute inner product */
double a, b, c, d, result;
result = a*b + c*d;
```

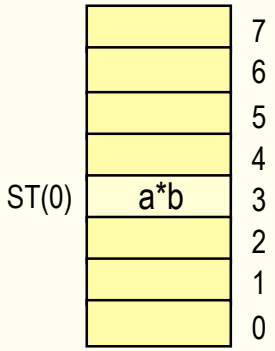
```
fld  a          /* Push a */
fmul b          /* a*b */
fld  c          /* Push c */
fmul d          /* c*d */
fadd st(1)      /* a*b + c*d */
fstp result     /* Pop result */
```



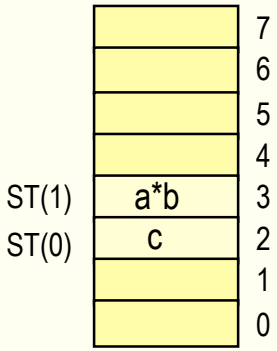
Initially, TOP is 4



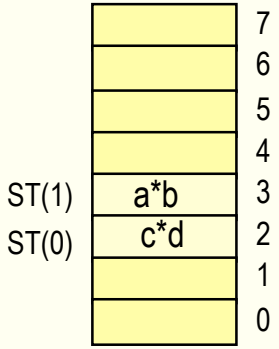
Load a



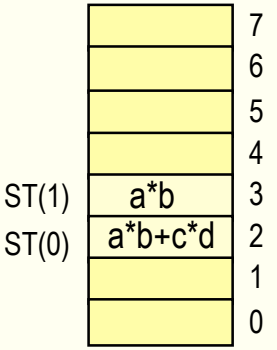
Multiply ST(0) with b



Load c



Multiply ST(0) with d



Add ST(0) and ST(1)

Floating-point instructions

- Floating-point instructions can be divided into the following groups
  - data transfer instructions
  - load constant instructions
  - basic arithmetic instructions
  - comparison instructions
  - transcedental instructions
  - FPU control instructions
- Most FP instructions have two operands
  - FP register or memory
  - ST(0) is often an implied operand
  - no immediate operands
- Operands can be floating-point, integer or packed BCD

Data transfer instructions

- Load operands from memory into ST(0)
  - FLD – Load Floating Point
  - FILD – Load Integer
- Store the value in ST(0) into memory
  - FST – Store Floating Point
  - FIST – Store Integer
  - FSTP – Store Floating Point and Pop
- Move values between FP register
  - FXCH – Exchange Register Contents
  - FCMOV*cc* – Conditional Move

Load constant instructions

- Instructions that push commonly used constant onto the top-of-stack
  - FLDZ – Load +0.0
  - FLDPI – Load π
  - FLDL2T – Load $\log_2 10$
  - FLDL2E – Load $\log_2 e$
  - FLDLG2 – Load $\log_{10} 2$
  - FLDLN2 – Load $\log_e 2$

Basic arithmetic instructions

- FADD / FADDP – Add Floating-Point (and Pop)
- FIADD – Add Integer to Floating point
- FSUB / FSUBP – Subtract Floating-Point (and Pop)
  - ST(0) ← ST(0)-ST(i)
- FSUBR – Reverse Subtract Floating Point
  - ST(0) ← ST(i)-ST(0)
- FMUL / FMULP – Multiply Floating-Point (and Pop)
- FDIV / FDIVP – Divide Floating-Point (and Pop)
- FCHS – Change Sign
- FABS – Absolute Value
- FSQRT – Square Root
- FRNDINT – Round To Integral Value

Comparing floating-point values

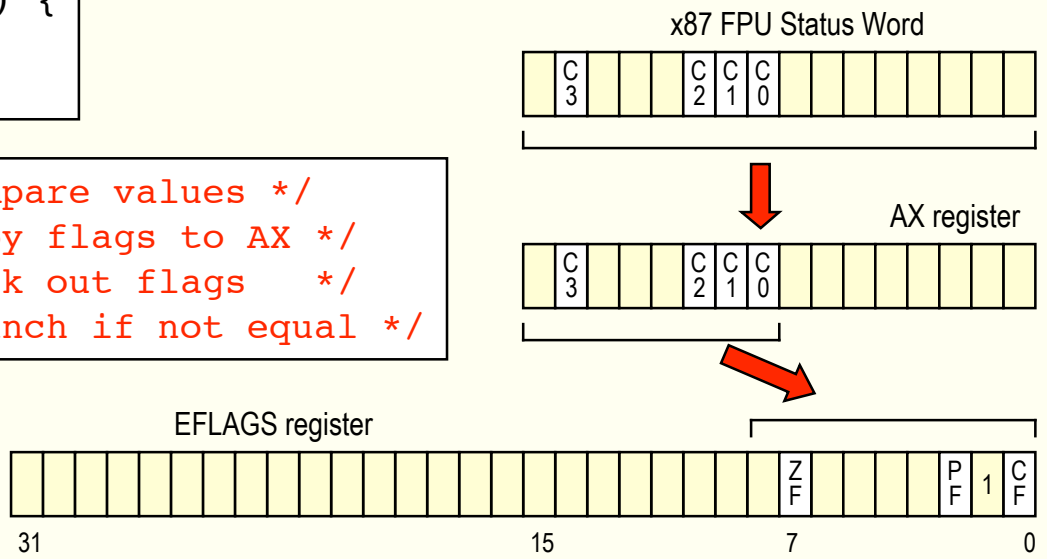
- Two mechanisms for comparing floating-point values and setting the status bits in EFLAGS register
  - used by conditional branch and conditional move instructions
- The old mechanism
  - floating-point compare instructions set the condition flags in the FP status register
  - the condition flags has to be copied into the status flags of the EFLAGS register
  - need three instructions for a comparison
- The new mechanism
  - introduced in the P6 microarchitecture (Pentium Pro and newer)
  - floating-point compare instructions directly set the condition flags in the EFLAGS register
  - need only one instruction for a comparison

Comparison instructions

- Old mechanism
  - FCOM / FCOMP / FCOMPP – compare ST(0) with source operand and set condition flags in FP status word (and Pop / Pop twice)
  - FTST – compare ST(0) with 0.0 and set condition flags in FP status word

```
double a, b;
   . . .
if (a > b) {
   . . .
}
```

```
fcomp   ST(0),ST(1) /* Compare values */
fstsw   AX          /* Copy flags to AX */
test    0x45,AH     /* Mask out flags   */
jne     L1          /* Branch if not equal */
```



Comparison instructions (cont.)

- New mechanism
  - FCOMI, FCOMIP – compare floating-point values and set EFLAGS (and Pop)

```
double a, b;
  . . .
if (a > b) {
  . . .
}
```

```
fcomi   ST(0),ST(1) /* Compare values */
jne     L1          /* Branch if not equal */
```

- gcc uses the old mecanism for comparing floating-point values

Transcedental instructions

- FSIN, FCOS – Sine, Cosine
- FPTAN, FPATAN – Tangent, Arctangent
- FYL2X – Logarithm
  - computes ST(1) ← ST(1) * $\log_2$ (ST(0)) and pops the register stack
- F2XM1 – Exponential
  - computes ST(0) ← $2^{ST(0)}$-1
- FSCALE – Scale ST(0) by ST(1)
  - computes ST(0) ← ST(0)*$2^{ST(1)}$

Control instructions

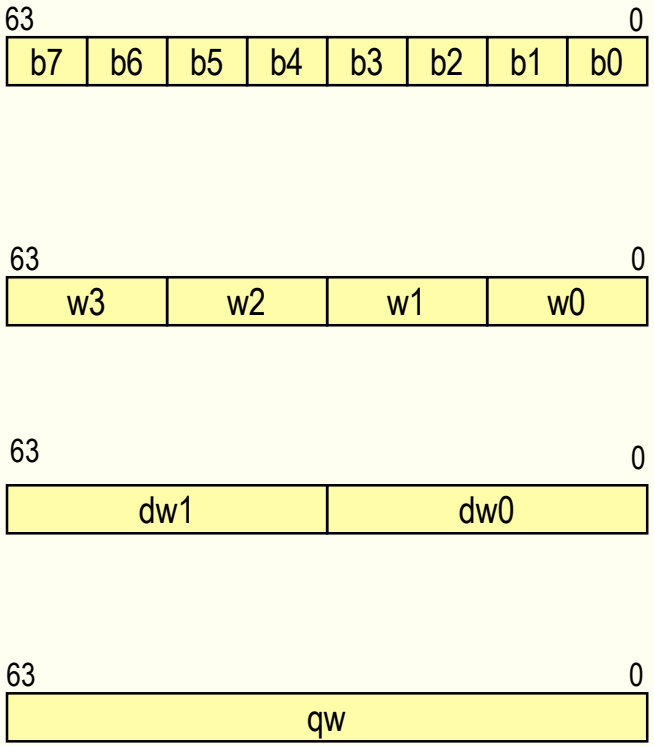
- FLDCW, FSTCW – Load / Store FPU Control Word
- FSAVE / FRSTOR – Save / Restore FPU State
- FINCSTP / FDECSTP – Increment / Decrement FPU Register Stack Pointer
- FFREE – Free FPU Register
- FNOP – FPU No Operation

MMX, SSE and SSE2

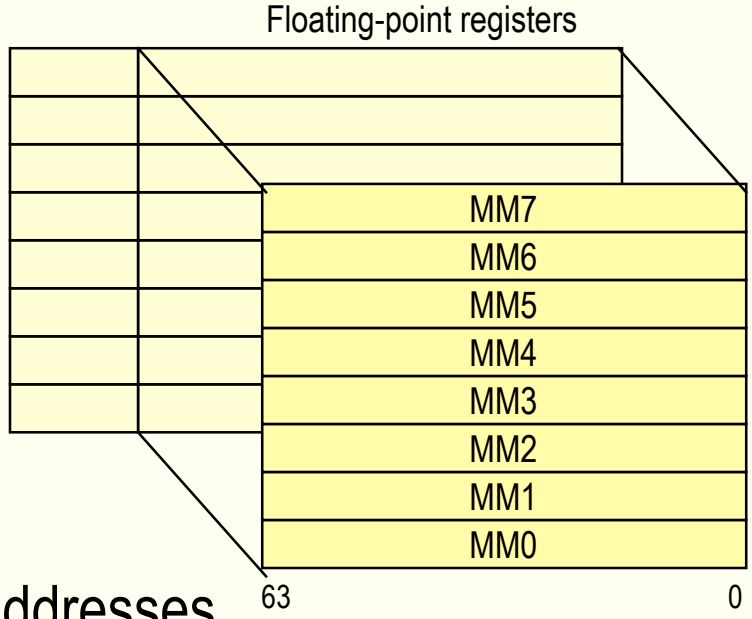
- Extensions to the instruction set for parallel SIMD operations on packed data
  - SIMD – Single Instruction Stream Multiple Data stream
- MMX – Multimedia Extensions
  - introduced in the Pentium processor
- SSE – Streaming SIMD Extension
  - introduced in Pentium III
- SSE2 – Streaming SIMD Extension 2
  - introduced in Pentium 4
- Designed to speed up multimedia and communication applications
  - graphics and image processing
  - video and audio processing
  - speech compression and recognition

MMX data types

- MMX instructions operate on 8, 16, 32 or 64-bit integer values, packed into a 64-bit field
- 4 MMX data types
  - packed byte
    8 bytes packed into a 64-bit quantity
  - packed word
    4 16-bit words packed into a 64-bit quantity
  - packed doubleword
    2 32-bit doublewords packed into a 64-bit quantity
  - quadword
    one 64-bit quantity
- Operates on integer values only



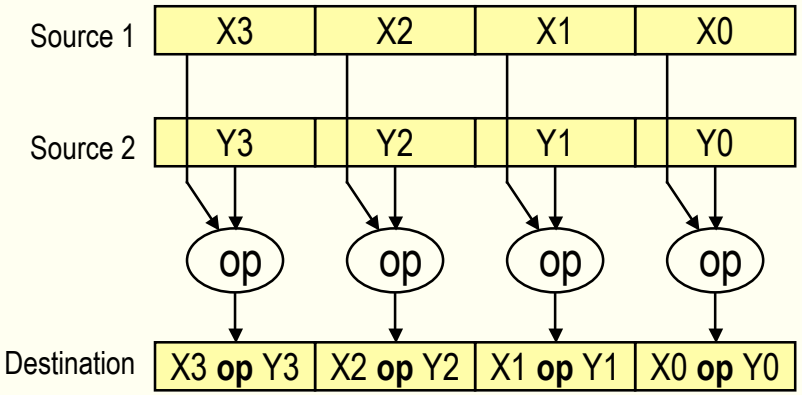
MMX registers



- 8 64-bit MMX registers
  - aliased to the x87 floating-point registers
  - no stack-organization
- The 32-bit general-purouse registers (EAX, EBX, ...) can also be used for operands and adresses
  - MMX registers can not hold memory addresses
- MMX registers have two access modes
  - 64-bit access mode
    - 64-bit memory access, transfer between MMX registers, most MMX operations
  - 32-bit access mode
    - 32-bit memory access, transfer between MMX and general-purpose registers, some unpack operations

MMX operation

- SIMD execution
  - performs the same operation in parallel on 2, 4 or 8 values
  - arithmetic and logical operations executed in parallel on the bytes, words or doublewords packed in a 64-bit MMX register
- Most MMX instructions have two operands
  - op dest source
  - destination is a MMX register
  - source is a MMX register or a memory location



Saturation and wraparound arithmetic

- Operations may produce results that are out of range
  - the result can not be represented in the format of the destination
- *Example*:
  - add two packed unsigned byte integers 154+205=359
  - the result can not be represented in 8 bits

```
 10011010
+11001101
---------
101100111
```

- Wraparound arithmetic
  - the result is truncated to the *N* least significant bits
  - carry or overflow bits are ignored
  - *example*: 154+205=103
- Saturation arithmetic
  - out of range results are limited to the smallest/largest value that can be represented
  - can have both signed and unsigned saturation
  - *example*: 154+205=255

Data ranges for saturation

- Results smaller than the lower limit is saturated to the lower limit
- Results larger than the upper limit is saturated to the upper limit
- Natural way of handling under/over-flow in many applications

Data type	Bits	Lower limit	Upper limit
Signed byte	8	-128	127
Unsigned byte	8	0	255
Signed word	16	-32768	32767
Unsigned word	16	0	65535

  - *Example*: color calculations, if a pixel becomes black, it remains black
- MMX instructions do not generate over/underflow exceptions or set over/underflow bits in the EFLAGS status register

MMX instructions

- MMX instructions have names composed of four fields
  - a prefix P – stands for packed
  - the operation, for example ADD, SUB or MUL
  - 1-2 characters specifying unsigned or signed saturated arithmetic
    - US – Unsigned Saturation
    - S – Signed Saturation
  - a suffix describing the data type
    - B – Packed Byte, 8 bytes
    - W – Packed Word, four 16-bit words
    - D – Packed Doubleword, two 32-bit double words
    - Q – Quadword, one single 64-bit quadword
- Example:
  - PADDB – Add Packed Byte
  - PADDSB – Add Packed Signed Byte Integers with Signed Saturation

MMX instructions

- MMX instructions can be grouped into the following categories:
  - data transfer
  - arithmetic
  - comparison
  - conversion
  - unpacking
  - logical
  - shift
  - empty MMX state instruction (EMMS)

Data transfer instructions

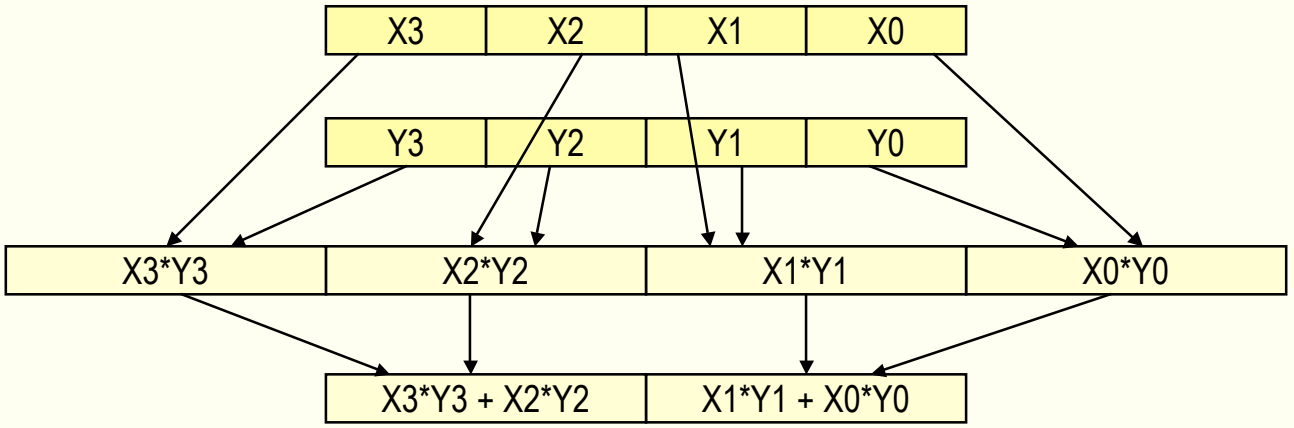
- MOVD – Move Doubleword
  - copies 32 bits of packed data
    - from memory to a MMX register (and vice versa), or
    - from a general-purpose register to a MMX register (and vice versa)
  - operates on the lower doubleword of a MMX register (bits 0-31)
- MOVQ – Move Quadword
  - copies 64 bits of packed data
    - from meory to a MMX register (and vice versa), or
    - between two MMX registers
- MOVD/MOVQ implements
  - register-to-register transfer
  - load from memory
  - store to memory

Arithmetic instructions

- Addition
  - PADDB, PADDW, PADDD – Add Packed Integers with Wraparound Arithmetic
  - PADDSB, PADDSW – Add Packed Signed Integers with Signed Saturation
  - PADDUSB, PADDUSW – Add Packed Unsigned Integers with Unsigned Saturation
- Subtraction
  - PSUBB, PSUBW, PSUBD – Wraparound arithmetic
  - PSUBSB, PSUBSW – Signed saturation
  - PSUBUSB, PSUBUSW – Unsigned saturation
- Multiplication
  - PMULHW – Multiply Packed Signed Integers and Store High Result
  - PMULLW – Multiply Packed Signed Integers and Store Low Result

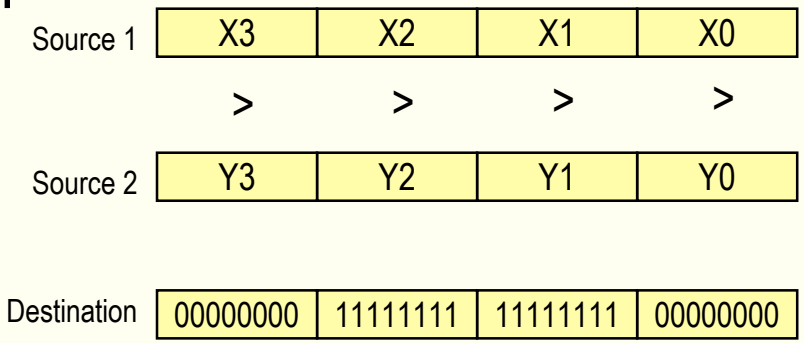
Arithmetic instructions (cont.)

- Multiply and add
  - PMADDWD – Multiply And Add Packed Integers
  - multiplies the signed word operands (16 bits)
  - produces 4 intermediate 32-bit products
  - the intermediate products are summed pairwise and produce two 32-bit doubleword results



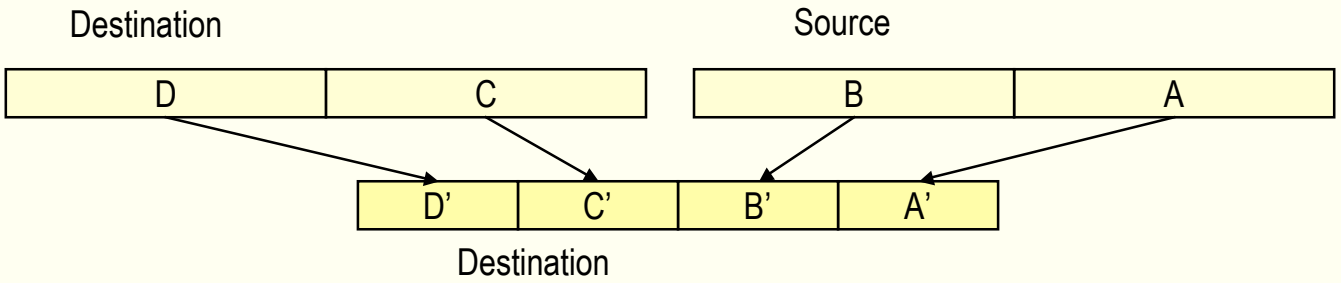
Comparison instructions

- Compare Packed Data for Equal
  - PCMPEQB, PCMPEQW, PCMPEQD
- Compare Packed Signed Integers for Greater Than
  - PCMPGTPB, PCMPGTPW, PCMPGTPD
- Compare the corresponding packed values
  - sets corresponding destination element to a mask of all ones (if comparison matches) or zeroes (if comparison does not match)
  - the masks can be used to implement conditional assignment
- Does not affect EFLAGS register



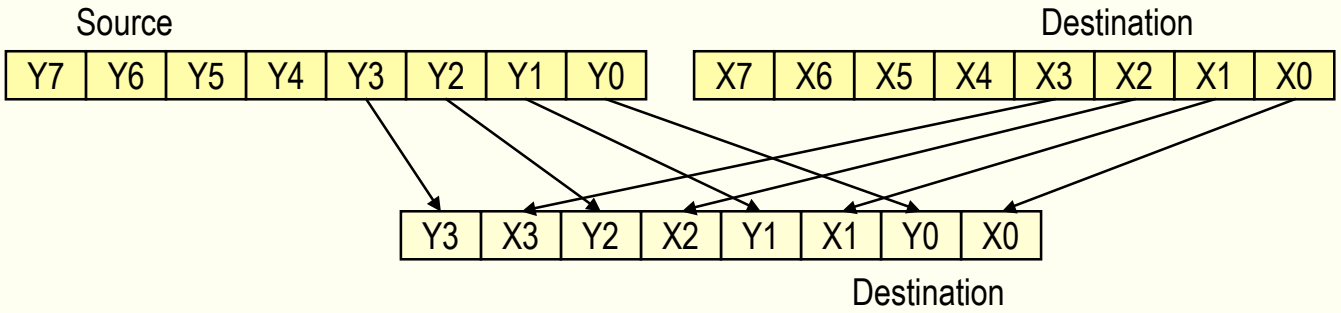
Conversion instruction

- PACKSSWB, PACKSSDW – Pack with Signed Saturation
- PACKUSWB – Pack with Unsigned Saturation
  - converts words (16 bits) to bytes (8 bits) with saturation
  - converts doublewords (32 bits) to words (16 bits) with saturation



Unpacking instructions

- PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ – Unpack and Interleave High Order Data
- PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ – Unpack and Interleave Low Order Data



Logical instructions

- PAND – Bitwise AND
- PANDN – AND NOT
- POR – OR
- PXOR – Exclusive OR

- Operate on a 64-bit quadwords

Shift instructions

- PSLLW, PSLLD, PSLLQ - Shift Packed Data Left Logical
- PSRLW, PSRLD, PSRLQ – Shift Packed Data Right Logical
- PSRAW, PSRAD – Shift Packed Data Right Arithmetic
  - shifts the destination elements the number of bits specified in the count operand

EMMS instruction

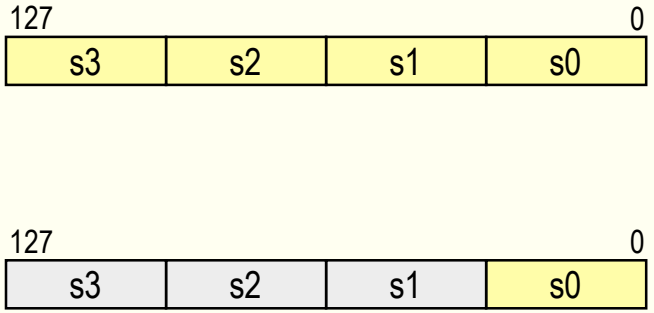
- Empty MMX State
  - sets all tags in the x87 FPU tag word to indicate empty registers
- Must be executed at the end of a MMX computation before floating-point operations
- Not needed when mixing MMX and SSE/SSE2 instructions

SSE

- Streaming SIMD Extension
  - introduced with the Pentium III processor
  - designed to speed up performance of advanced 2D and 3D graphics, motion video, videoconferencing, image processing, speech recognition, ...
- Parallel operations on packed single precision floating-point values
  - 128-bit packed single precision floating point data type
  - four IEEE 32-bit floating point values packed into a 128-bit field
- Introduces also some extensions to MMX
- Operand of SSE instructions must be aligned in memory on 16-byte boundaries

SSE instructions

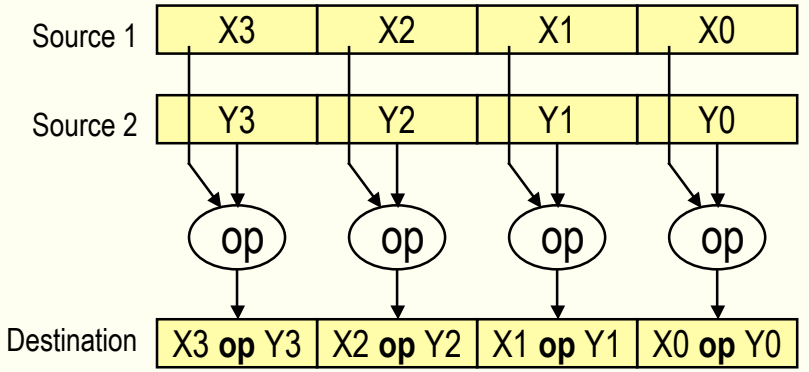
- Adds 70 new instructions to the instruction set
  - 50 for SIMD floating-point operations
  - 12 for SIMD integer operations
  - 8 for cache control
- Operates on packed and scalar single precision floating-point instructions
  - operations on packed 32-bit floating-point values
  - operations on a scalar 32-bit floating-point value (the 32 LSB)



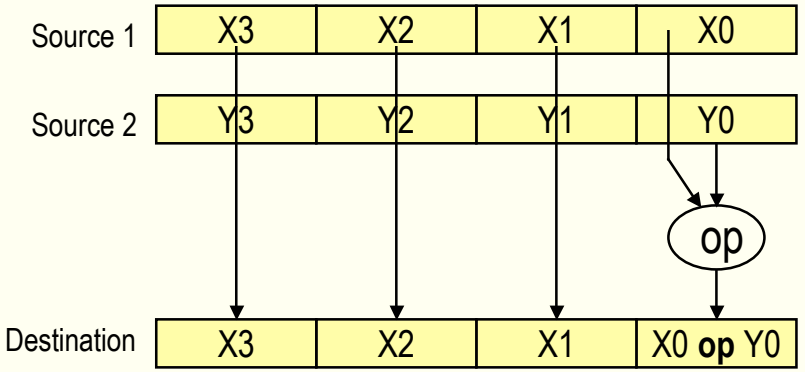
- 64-bit SIMD integer instructions
  - extension to MMX
  - operations on packed integer values stored in MMX registers

Packed and scalar operations

- SSE supports both packed and scalar operations on 32-bit floating-point values
- Packed operations applies the operation in parallel on all four values in a 128-bit data item
  - similar to MMX operation

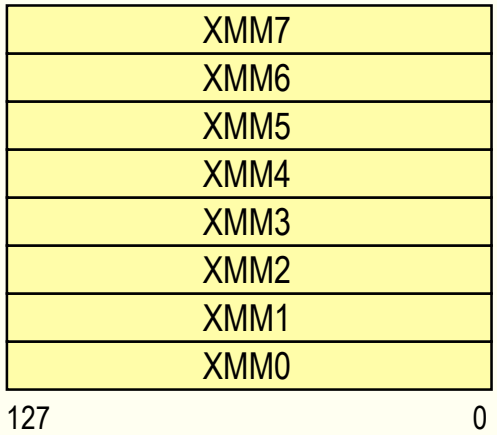


- Scalar operations operates only on the least significant 32 bits



XMM registers

- The MMX technology introduces 8 new 128-bit registers XMM0 – XMM7
  - independent of the general purpose and FPU/MMX registers
  - can mix MMX and SSE instructions
- XMM registers can be accessed in 32-bit, 64-bit or 128-bit mode
  - only for operations on data, not addresses
- MXCSR control and status register, 32 bit
  - flag and mask bits for floating-point exceptions
  - rounding control bits
  - flush-to-zero bit
  - denormals-are-zero bit



SSE instructions

- SSE instructions are divided into four types
- Packed and scalar single-precision floating point operations
  - operates on 128-bit data entities
- 64-bit integer operations
  - MMX operations
- State manegement intructions
  - load and save state of the MXCSR control register
- Cache control, prefetch and memory ordering instructions
  - instructions to control stores to / loads from memory
  - support for streaming data to/from memory without storing it in cache

Temporal and non temporal data

- Temporal data
  - data that will be used more than once in the program execution
  - should be accessed through the cache to make use of the temporal locality
- Non-temporal data
  - data that will not be reused in the program execution
  - if non-temporal data is accessed through the cache it will replace temporal data – called *cache pollution*
  - can be accessed from memory without going through the cache using non-temporal prefetching and write-combining
- Media processing applications often have large amounts of non-temporal data
  - streaming data

Cacheability control and prefetching

- Data can be read into the cache in advance using a *prefetch* operation
- Three levels of prefetch for temporal data
  - PREFETCH0 – fetch data into all cache levels
  - PREFETCH1 – fetch data into L2 cache (and higher)
  - PREFETCH2 – fetch data into L3 cache
- Prefetching of non-temporal data with PREFETCHNTA
  - fetch data into an internal buffer
  - data is not stored in cache
- Non-temporal data can be written without going through the cache
  - uses write-combining: data is combined into larger blocks before written to main memory
  - gives less control of the order of writes to memory

SSE2

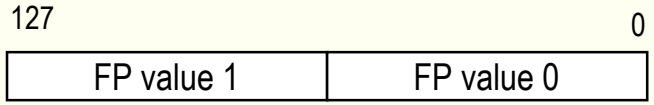
- Streaming SIMD Extension 2
  - introduced in the Pentium 4 processor
  - designed to speed up performance of advanced 3D graphics, video encoding/decodeing, speech recognition, E-commerce and Internet, scientific and engineering applications
- Extends MMX and SSE with support for
  - 128-bit packed double precision floating point-values
  - 128-bit packed integer values
- Adds over 70 new instructions to the instruction set
- Operates on 128-bit entities
  - data must be aligned on 16-bit boundaries when stored in memory
  - special instruction to access unaligned data

Compatibility with SSE and MMX operation

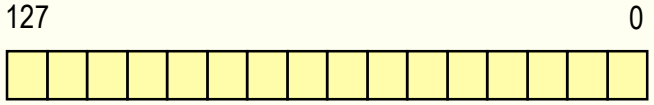
- The SSE2 extension is an enhancement of the SSE extension
  - no new registers or processor state
  - new instructions which operate on a wider variety of packed floating-point and integer data
- Same registers for SIMD operations as in SSE
  - eight 128-bit registers, XMM0 – XMM7
- SSE2 instructions can be intermixed with SSE and MMX/FPU instructions
  - same registers for SSE and SSE2 execution
  - separate set of registers for FPU/MMX instructions

SSE2 data types

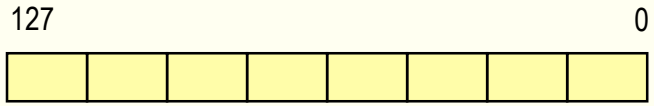
- Packed double precision floating point
  - 2 IEEE double precision floating-point values



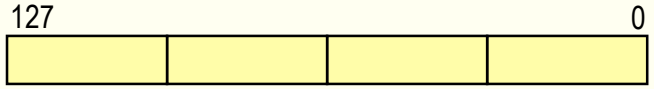
- Packed byte integer
  - 16 byte integers (8 bits)



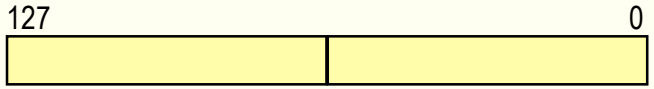
- Packed word integer
  - 8 word integers (16 bits)



- Packed doubleword integer
  - 4 doubleword integers (32 bits)



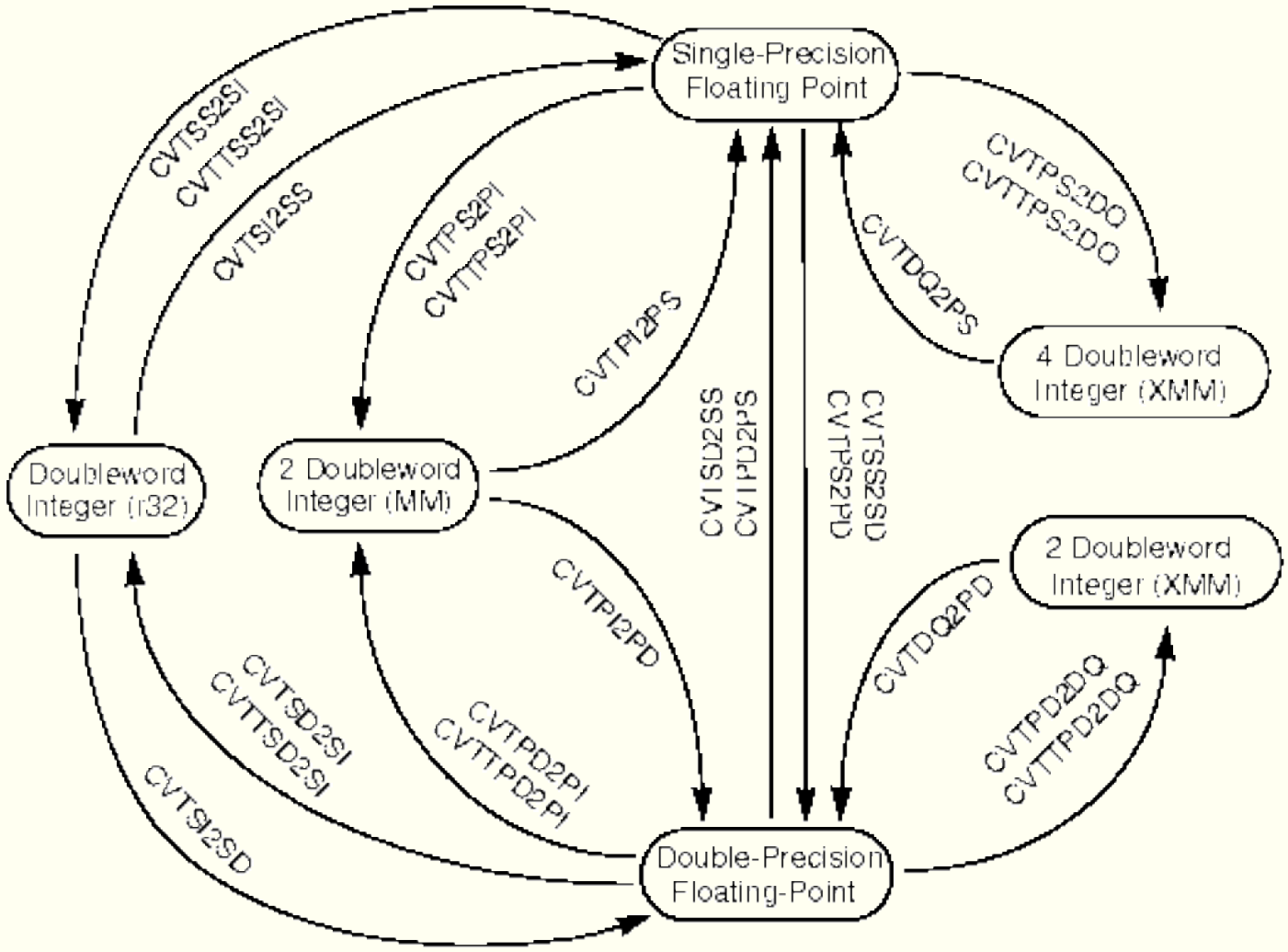
- Packed quadword integer
  - 2 quadword integers (64 bits)



SSE2 instructions

- Operations on packed double-precision data has the suffix PD
  - *examples*: MOVAPD, ADDPD, MULPD, MAXPD, ANDPD
- Operations on scalar double-precision data has the suffix SD
  - *examples*: MOVSD, ADDSD, MULSD, MINSD
- Conversion instructions
  - between double precision and single precision floating-point
  - between double precision floating-point and doubleword integer
  - between single precision floating-point and doubleword integer
- Integer SIMD operations
  - both 64-bit and 128-bit packed integer data
  - 64-bit packed data uses the MMX register
  - 128-bit data uses the XMM registers
  - instructions to move data between MMX and XMM registers

Conversion between packed data types



Programming with MMX and SSE

- Automatic vectorization
  - let the compiler do all the work, just turn on a compiler switch
  - easy to program, no changes to the program code
  - only loops are vectorize
  - does not guarantee any performance improvement
    - has no effect if the compiler can not analyze the code and find opportunities for SIMD operation
  - requires a vectorizing compiler
- C++ class data types
  - C++ classes that define an abstraction for the MMX/SSE datatypes
  - easy to program, does not require in-depth konwledge of MMX/SSE
  - guarantees a performance improvement
  - can not access all possible instructions
  - can not do explicit instruction scheduling

Programming with MMX and SSE (cont.)

- Compiler intrisincs
  - functions that perform the same operations as the corresponding assembly language instructions
  - gives access to all MMX and SSE instructions
  - can use variable names instead of registers
  - requires a detailed knowledge of MMX/SSE operation
- Assembly language
  - gives full control over the instruction execution
  - very good possibilities to arrange instructions for efficient execution
  - difficult to program, requires detailed knowledge of MMX/SSE operations and assembly language programming

Example: summing an array of integers

- Simple function that sums all elements in an array of integer values and returns the sum

```
int SumArray(int *buf, int N)
{
  int i, sum=0;
  for (i=0; i<N; i++)
       sum += buf[i];
  return sum;
}
```

- To compile for automatic vectorization with the Intel compiler use the switch -QxW
  - the compiler prints a message about vectorized loops

```
program.cpp (42) : (col. 2) remark: LOOP WAS VECTORIZED
```

C++ class libraries for SIMD operation

- C++ classes defining MMX and SSE data types
  - overloads the operations +, -, *, / etc.
- Integer data types
  - I8vec8, I8vec16
  - I16vec4, I16vec8
  - I32vec2, I32vec4
  - I64vec1, I64vec2
  - I128vec1
- Single precision floating-point data types
  - F32vec1, F32vec4
- Double precision floating-point data types
  - F64vec2

```
int SumArray(int *buf, int N)
{
  int i;
  I32vec4 *vec4 = (I32vec4 *)buf;
  I32vec4 sum(0,0,0,0);
  for (i=0; i<N/4; i++)
       sum += vec4[i];
  return sum[0]+sum[1]+sum[2]+sum[3];
}
```

C / C++ compiler intrisincs

- Functions or macros containing inline assembly code for MMX/SSE operations
  - allows the programmer to use C / C++ function calls and variables
- Defines a C function for each MMX/SSE instruction
  - there are also intrisinc functions composed of several MMX/SSE instructions
- Defines data types to represent packed integer and floating-point values
  - __m64 represents the contents of a 64-bit MMX register (8, 16 or 32 bit packed integers)
  - __m128 represents 4 packed single precision floating-point values
  - __m128d represents 2 packed double precision floating-point values
  - __m128i represents packed integer values (8, 16, 32 or 64-bit)

C intrisincs

- The code specifies exactly which operations to use
  - register allocation and instruction scheduling is left to the compiler

```
int SumArray(int *buf, int N)
{
  int i;
  __m128i *vec128 = (__m128i *)buf;
  __m128i sum;
  sum = _mm_sub_epi32(sum,sum); // Set to zero
  for (i=0; i<N/4; i++)
       sum = _mm_add_epi32(sum,vec128[i]);

  sum = _mm_add_epi32(sum, _mm_srli_si128(sum,8));
  sum = _mm_add_epi32(sum, _mm_srli_si128(sum,4));
  return _mmcvtsi128_si32(sum);
}
```

Assembly language

- Use inline assembly code
  - for instance in a C program
- Can arrange instructions to avoid stalls

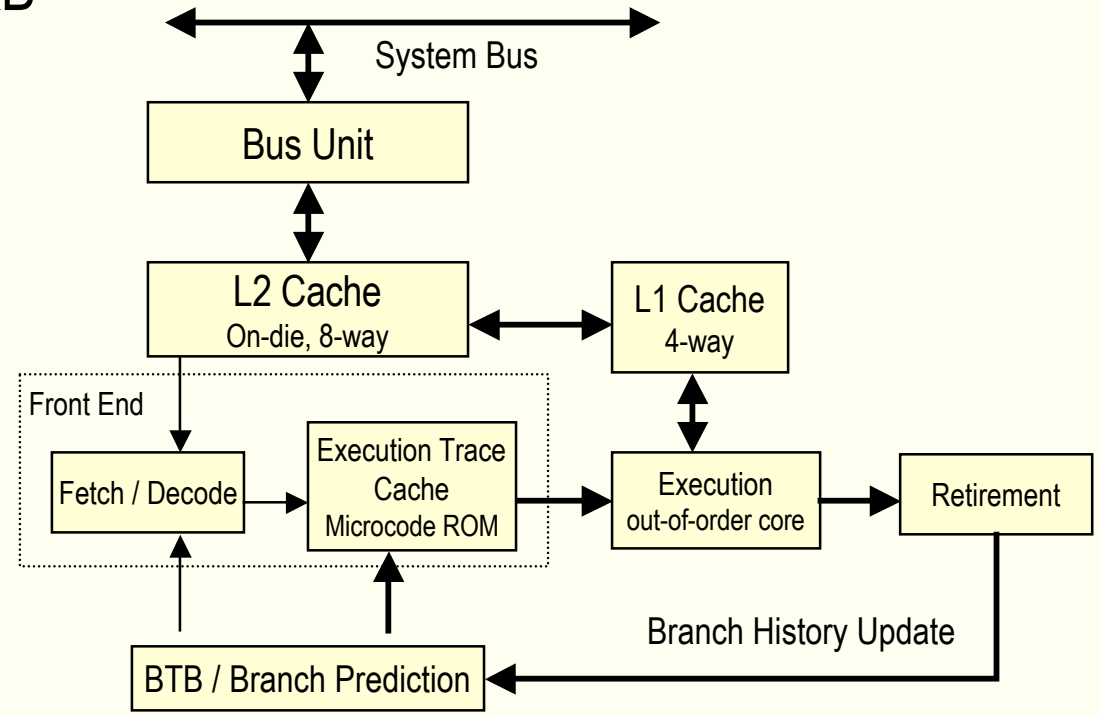
```
int SumArray(int *buf, int N)
{
_asm{
    mov  ecx, 0 ; loop counter
    mov  esi, buf
    pxor xmm0,xmm0 ; zero sum
loop:
    paddd xmm0, [esi+ecx*4]
    add  ecx, 4
    cmp ecx, N ; done ?
    jnz loop
    movdqa xmm1, xmm0
    psrldq xmm1, 8
    padd xmm0,xmm1
    movdqa xmm1,xmm0
    psrldq xmm0,xmm1
    movd eax, xmm0 ; store result
   }
}
```

Intel Pentium 4

- Based on the Intel NetBurst microarchitecture
  - Pentium II and Pentium III are based on th P6 microarchitecture
- Decoupled CISC/RISC architecture
  - IA-32 instruction set, CISC
  - translated to RISC micro-operations (□ops), which are executed by the RISC core
- Deep pipeline
  - designed to run at very high clock frequencies
    - introduced at 1.5 GHz
    - currently at 3.2 GHz
  - different parts of the chip run at different clock frequencies
- Efficient execution of the most common instructions
- SSE2 extension

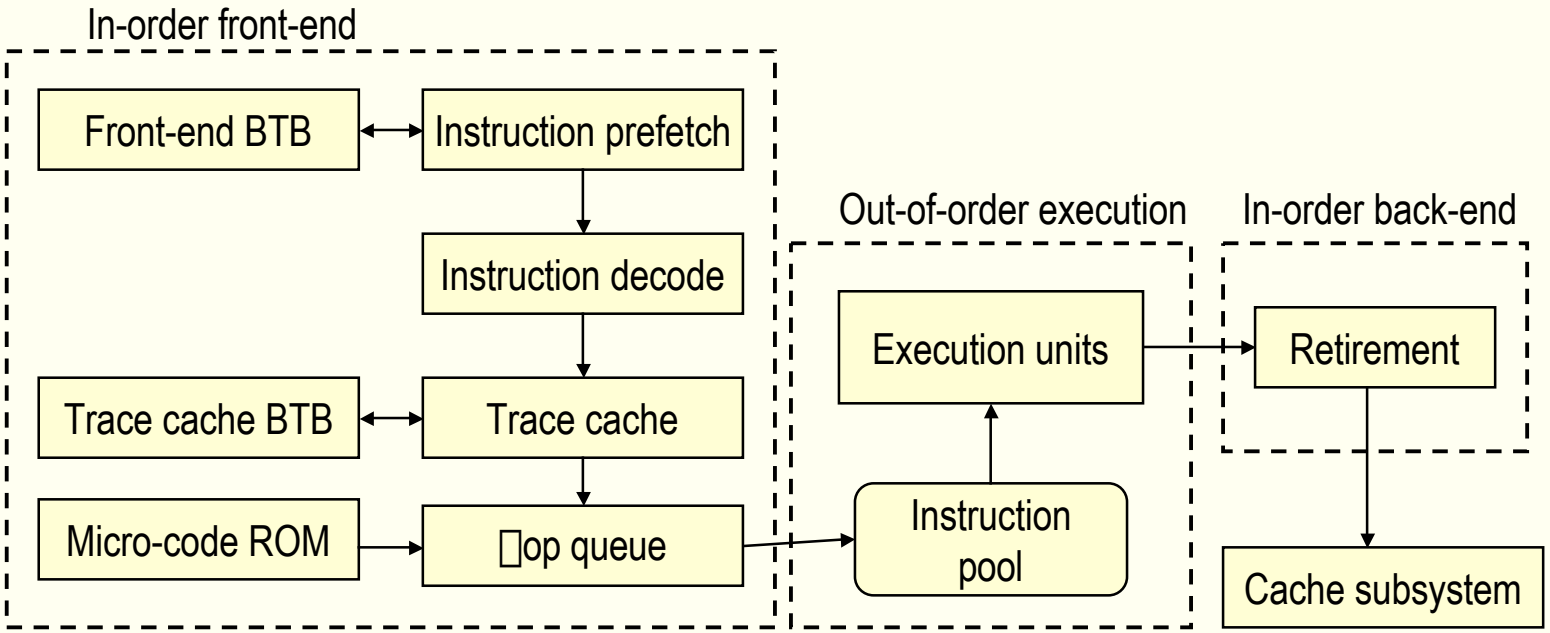
NetBurst microarchitecture

- Cache
  - execution trace cache, 12K □ops
  - L1 data cache, 8 KB, 2 cycle latency
  - L2 cache on-die, 512 KB
    7 cycle latency
- 20-stage pipeline, supports high clock frequencies
  - ALU runs twice the processor clock frequency
  - quad-pumped system bus interface



Pipeline organization

- The pipeline consists of three sections
  - in-order issue front-end with a execution trace cache
  - out-of-order superscalar exection core with a very deep out-of-order speculative execution engine
  - in-order retirement



Front-end pipeline

- Designed to improve the instruction decoding capabilities
  - improves the time to decode fetched instructions
  - avoids problems with wasted decode bandwidth caused by branches and branch targets in the middle of cache lines
- Basic functions of the front-end
  - prefetch IA-32 instructions that are likely to be executed
  - fetch instructions that have not been prefetched
  - decode IA-32 instructions into □ops
  - generate microcode for complex instructions
  - store decoded □ops in the execution trace cache
  - deliver decoded instructions from the execution trace cache to the execution core
  - predict branches

Prefetching

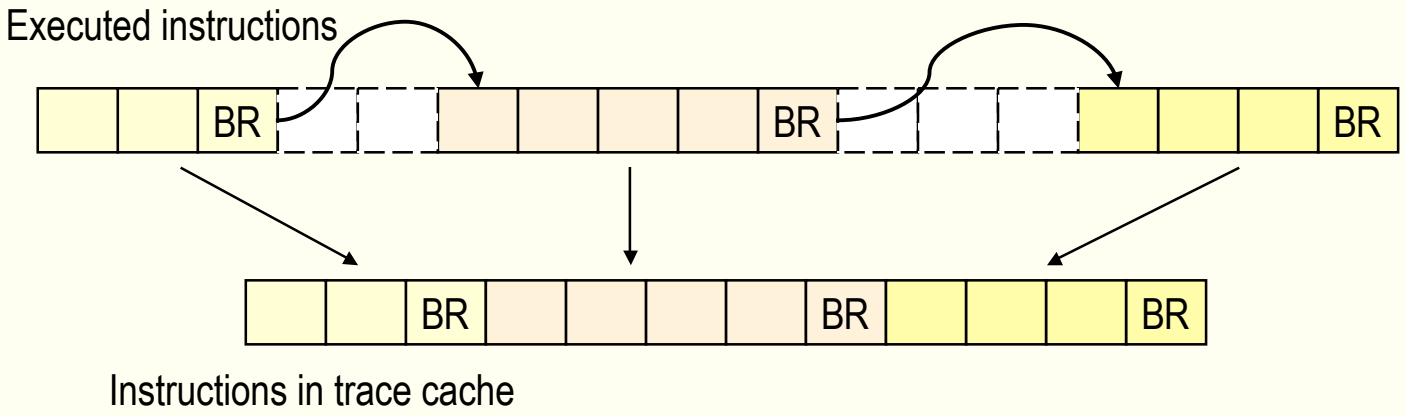
- Automatic data prefetch
  - hardware that auomatically prefetchs data into L2 cache
  - based on previous access patterns
  - tries to fetch data 2 cache lines ahead of current access location (but only within the same 4 KB page)
- Software prefetch
  - prefetch instructions, only for data access
  - hint to the hardware to bring in a cache line
- Instructions are automatically prefetced from the predicted execution path into an instruction buffer
  - fetched from L2 cache into a buffer in the instruction decoder

Instruction decoding

- IA-32 machine instructions are of variable length
  - large number of options for most instructions
- Decoded to uniform-length micro-operations
  - load/store architecture
- IA-32 instructions can be decoded into one or more □ops
  - if more than 4 □ops are needed, the instruction is decoded from the microcode ROM
- Decoded □ops are stored in program order in the execution trace cache
  - do not need to be decoded the next time the same code is executed

Execution trace cache

- Instruction cache storing decoded instructions
  - 12K □ops, 8-way set-associative
- Stores fetched and decoded instructions
  - built into sequences of □ops called *traces,* six □ops per trace line
  - contains □ops generated from the predicted execution path
  - instructions that are branched over in the execution will not be in the trace cache



Execution trace cache (cont.)

- The trace cache can deliver 3 instructions each clock tick to the out-of-order execution logic
- Most instructions are fetched and executed from the trace cache
  - only when there is a trace cache miss does the instructions have to be fetched from L2 cache
  - reduces the amount of work for the instruction decoder
- The trace cache has an own branch predictor
  - predicts branches within the trace cache

Branch prediction

- Branch target buffer, 4K entries
  - contains both branch history and branch target addresses
- Return address stack, 16-entries
  - contains return addresses for procedure calls
- Trace cache and instruction translation have co-operating branch prediction
  - branch targets are predicted based on information in BTB, RAS or using static prediction
  - branch target code is fetched from trace cache if it is there, otherwise from the memory hierarchy
- “Highly advanced branch prediction algorithm”
  - 33% less misprediction compared to the P6 mictoarchitecture

Branch prediction (cont.)

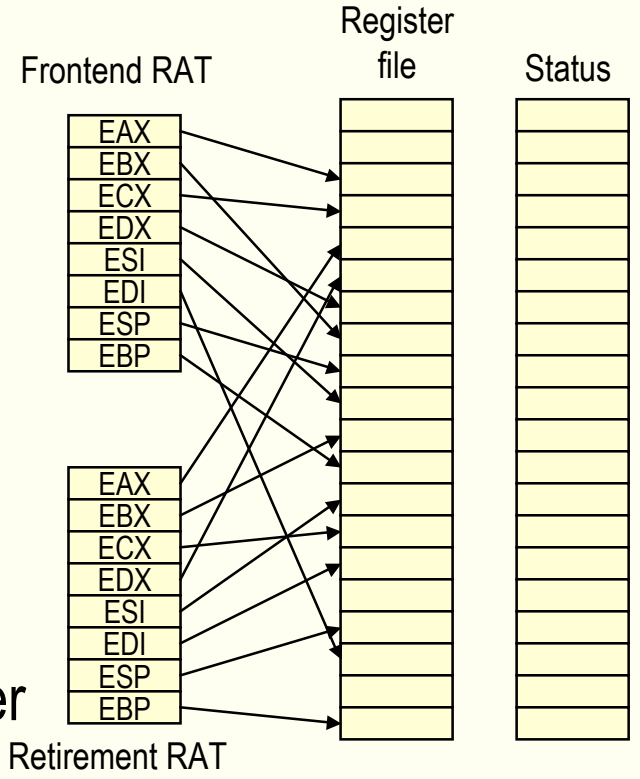
- Branch hints
  - prefix to conditional branch instructions
  - used to help the branch prediction and decoder to build traces
  - overrides static prediction, but not dynamic
- Branch hints have no effect on decoded instructions that already are in the trace cache
  - only assist the branch prediction and the decodeer to build correct traces
- Typical delay for a mispredicted branch is the depth of the pipeline
  - 20 clock ticks

Execution core

- Up to 126 instructions, 48 loads and 24 stores can be in fligtht at the same time
- Can dispatch up to 6 □ops per cycle
  - exceeds the capacity of the decoder and retirement unit
- Basic integer (ALU) operations execute in 1/2 clock cycle
- Many floating-point instructions can start every 2 cycles
- Floating-point divide and square root are not pipelined
  - *Example*: FP double precision divide
    latency = throughput = 38 clock cycles
- □ops are issued through four ports to 7 functional units
  - some ports can issue 2 □ops per clock cycle

Register renaming

- Renames the eight logical IA-32 registers to a 128-entry physical register file
  - uses a Register Alias Table (RAT) to store the renaming
- Similar register renaming for both integer and FP/MMX/SSE registers
- RAT points to the entry in the register file holding the current version of each register
  - the status stores information about the completion of the □op
- Load and stores are renamed similarly
  - uses a load/store buffer instead of a register

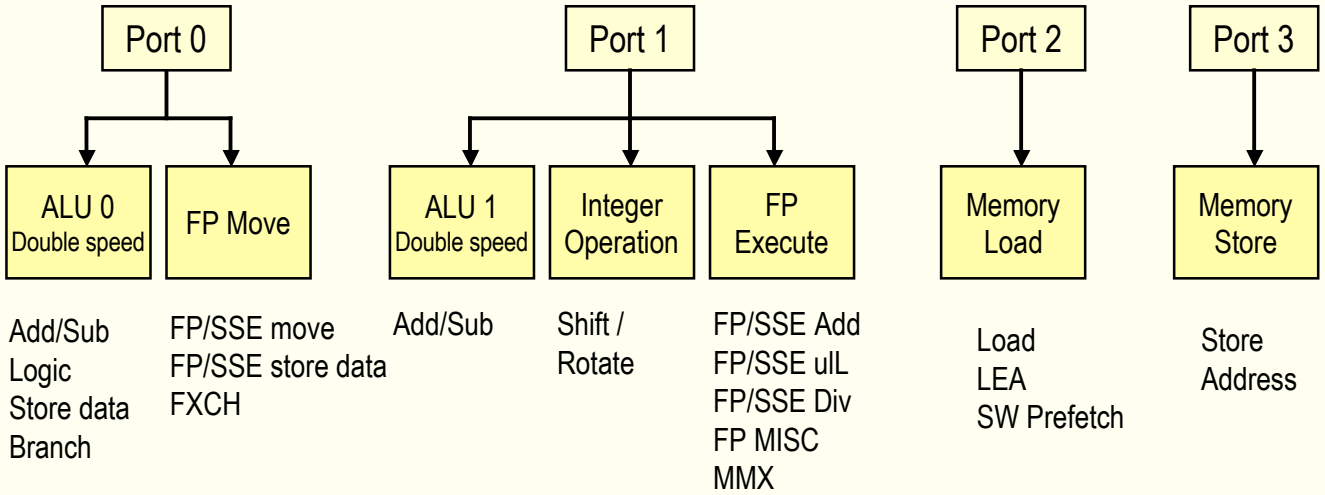


□op scheduling

- The □op shedulers determine when an operation is ready to be executed
  - when all its input operands are ready and a suitable execution unit is available
- The scedulers are connected to four dispatch ports
  - two execution unit ports
  - one load port
  - one store port
- The scedulers dispatch □ops to one of the ports depending on the type of the operation
  - can dispatch up to 6 □ops in a clock cycle
  - some ports can dispatch two operations in one clock cycle, operate on double clock cycle

Dispatch ports and execution units

- Port 0 can issue
  - 1 FP move or 1 integer ALU □op
  - +1 integer ALU □op
- Port 1 can issue
  - 1 FP □op or 1 integer □op or 1 integer ALU □op
  - +1 integer ALU □op
- Port 2 can issue 1 load □op
- Port 3 can issue 1 store adress □op



Loads and stores

- Out-of-order memory operations
  - loads can be executed speculatively
  - stores are always executed in program order
- Separate rename registers for memory access
  - 48 load buffers and 24 store buffers
  - hold the load/store □op and address information
  - one load and one store can be issued every clock cycle
- Store forwarding
  - a load from a memory location that is waiting to be stored does not have to wait for the memory operation to complete
  - data is forwarded from the store buffer to the load buffer
- Write combining
  - multiple stores to the same cache line are combined into one unit
  - 6 write-combine buffers

Retirement

- Receives result of executed μops and updates the processor state in program order
  - original program order is stored in the reorder buffer
  - up to 3 μops may be retired per clock cycle
- Sends updated branch target information to the branch target buffer
  - result of conditional branches are not known before the instruction is retired
  - recovers from branch misprediction

Cache organization

- Execution trace cache
  - 12 K ops, 8-way set associative
- L1 data cache, 8 KB, 4-way set associative, write through
  - fast, 2 clock cycle latency
  - 64 byte cache line size
- Unified L2 cache, 256 or 512 KB, 8-way set associative, write back
  - cache interface is 32 bytes = 256 bits
  - transfers data on each clock cycle
  - 128 byte cache line size (two 64 byte sectors)
  - latency 7 clock cycles, can start next transfer after 2 clock cycles
  - hardware prefetch
    - fetches data 256 bytes ahead of the current data access location
- Also support for L3 cache

Latency and throughput

- Latency
  - the number of clock cycles required for the execution core to complete the execution of an IA-32 instruction
- Throughput
  - the number of clock cycles the execution core has to wait before an issue port is ready to accept the same instruction again
- If throughput is less than latency than the execution unit is pipelined
  - can accept the following instruction before the previous one has completed
- Different instructions have different latency and throughtput

Instruction latency and throughput

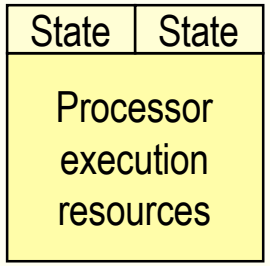
- Integer operations
  - latency 0.5–4, throuhput 0.5–2
  - mul, div has latency 15–70, throughput 5–40
- Floating-point operations
  - latency 2–7, throughput 1–2
  - division and square root have latency 23–43, throughput 23–43
  - sin, cos, tan, arctan have latency 150–250, throughput 130–170
- MMX operations
  - latency 2–6, throughput 1
- Integer SSE instructions
  - latency 2–8, throughput 1–2
- Single-precision floating-point SSE instructions
  - latency 4–10, throughput 2–4
  - div, square root has latency 32, throuhput 32

Instruction latency and throughput (cont.)

- Integer SSE2 instructions
  - latency 2–10, throughput 1–2
- Single-precision floating-point SSE2 instructions
  - latency 4-10, throughput 2–4
  - packed division has latency 39, throughput 18
  - packed square root has latency 39, throughput 29
- Double-precision floating-point SSE2 instructions
  - latency 4–10, throughput 2–4
  - packed division has latency 69, throughput 32
  - packed square root has latency 69, throughput 58

Hyper-threading microarchitecture

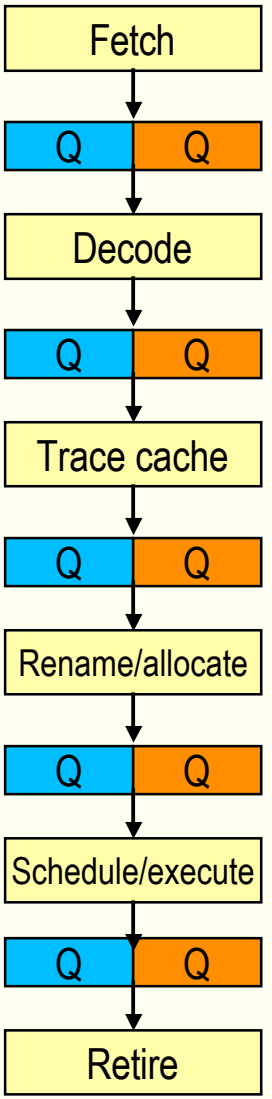
- Hyper-threading
  - simultaneous multi-threading
  - a single processor appears as two logical processors
  - both logical processors share the same physical execution resources
  - the architectural state is duplicated (register, program counter,status flags, ...)



- Can scedule two simultaneously executing threads on the processor
  - instructions from both threads execute simultaneously
  - if one thread has to wait, the other can proceed
- Makes more efficient use of the physical execution resources
  - uses task-level parallelism to increase the utilization of the execution resources

Pipeline organization

- Small dia area cost for implementing HT
  - about 5% of the total die area
  - most resources are shared, only a few are duplicated
- When one thread is stalled, the other can continue executing
  - one thread can not reserve all execution resources
  - shared resources are either partitioned between the threads or there is a limit on the amount of resources one logical process can use
- If only one thread is running it has full access to all execution resources
  - runs with the same speed as on a processor without HT



Processor resources

- Replicated resources
  - the resources needed to store information about the state of the two logical processors
    - registers, instruction pointer, control registers, register renaming tables, interrupt controllers
  - some resources are also replicated for efficiency reasons
    - instruction translation lookaside buffer, streaming instruction buffers, return address stack
- Partitioned resources
  - shared, but the use is limited to half of the entries
  - the instruction buffers between major pipeline stages: the queue after the trace cache, queues after register renaming, reorder buffers, load- and store buffers
- Shared resources
  - most resources are shared
  - execution units, branch prediction, caches, bus interface, ...

Instruction fetch

- Two sets of instruction pointers, one for each logical process
  - instruction fetch alternates between the logical processors each clock cycle, one cache line at a time
  - if one of the logical processes is stalled, the other gets full instruction fetch bandwidth
- If the next instruction is not in the trace cache it is fetched from L2 cache
  - the hardware resources for fetching data and instructions from L2 cache are duplicated
  - fetched instructions are placed in a streaming buffer, one for each logical processor
  - decoded and placed in the trace cache

Instruction decode

- Both logical procesors share the same decoding logic
  - alternates accesses for instructions to decode between the two streaming buffers
  - decodes several instructions for each logical processor before switching to the other
- Both logical processors share the microcode ROM
  - one pointer into microcode ROM for each logical process
  - alternates accesses to the microcode ROM each clock cycle
- If only one logical processor needs the decode logic it gets the full decode bandwidth

Trace cache

- The execution trace cache is shared between the two logical processors
  - access alternates every clock cycle
  - the trace cache entries also include information about to which thread it belongs
- One logical processor can have more entries in the trace cache than the other
  - if one thread is stalled for a long time, the other can fill the whole trace cache
- Decoded instructions are placed in a □op queue
  - decouples the front-end from the out-of-order execution engine
  - the □op queue is partitioned: each logical processor has half of the entries

Branch prediction

- Branch history buffer is partitioned between both logical processors
  - entries are tagged with the logical processor ID
- The global branch pattern history array is shared
- Return stack buffer is duplicated
  - 16 enties per logical processor

Allocation

- The allocation stage takes □ops from the queue and allocates the resources needed to execute the □op
  - register reorder buffers
  - integer- and floating point physical registers
  - load- and store buffers
- Each logical processor can use at most half of the register reorder buffers, load- and store buffers
- The allocator alternates between □ops from the logical processors every clock cycle
  - if one logical processors has used its full limit of some resource, it is stalled
  - if there are only □ops from only one logical processor it has to execute using only half of the available resources

Register renaming

- Register renaming is done at the same time as the resource allocation
- One Register Alias Table for each logical processor
  - stores the current mapping of the 8 architectural registers to the 128 physical registers
- After resource allocation and register renaming the □ops are placed in one of two queues for sceduling
  - memory instruction queue for loads and stores
  - general instruction queue for all other operations
- Both queues are partitioned so that each logical processor can use at most half of the entries

Instruction scheduling

- The memory instruction queue and the general instruction queue sends □ops to the scedulers
  - alternates between □ops from the logical processors every clock cycle
- The schedulers dispatches □ops to the different execution units when the inputs are ready and a suitable unit is free
  - selects □ops to dispatch from queues of 8–12 □ops
  - number of entries in each queue for a logical processor is limited
  - dispatches ready □ops regardless of which logical processor they belong to
- At most six □ops dispatched each clock cycle
  - can for instance dispatch two □ops from one logical processor and two □ops from the othe logical processor in the same clock cycle

Execution and retirement

- The execution units do not need to know to which logical processor a □op belongs
  - source and destination registers have already been renamed
  - the □ops access the physical register file, which is shared
  - results are written back to the physical register file
- Executed □ops are placed in the re-order buffer
  - re-order buffer is partitioned so that each logical processor can use at most half of the entries
- Executed □ops are retired in program order for each logical processor
  - alternates between the two logical processors

Single-task and multi-task modes

- A processor with hyper-threading can execute in two modes:
  - Single-Task mode (ST)
  - Multi-Task mode (MT)
- In MT mode there are two active logical processors
  - some execution resources are partitioned as decribed
- In ST mode one of the two logical processors are active and the other one is inactive
  - resources that were partitioned in MT mode are recombined
- Transition from MT mode to ST mode by executing a HALT instruction on one of the logical processors
  - an interrupt sent to the halted logical processor resumes its execution and places the processor in MT mode
  - the operating system is responsible for controlling transitions between ST and MT mode