Computer Architecture Classifications (Flynn 1966)

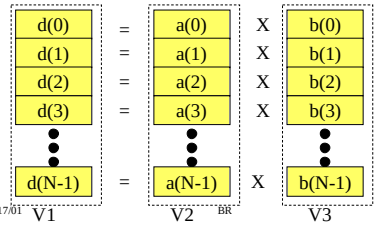
- SISD - Single Instruction stream, Single Data stream
  - classical uniprocessor
- SIMD - Single Instruction stream, Multiple Data streams
  - Vector machines
- MISD - Multiple Instruction streams, Single Data Streams
  - No good examples, not very useful
- MIMD - Multiple Instruction streams, Multiple Data Streams
  - Most interesting, many examples

SIMD (Vector Machines)

- Simple idea – apply the same operation to multiple sets of data
  - called a Vector operation

mul v1, v2, v3

(v1, v2, v3 are vector registers each contain N sets of data)



Supercomputers

- The term 'SuperComputer' used to be applied to Vector machines
  - Cray Research created the first supercomputer (CRAY-1) in 1976
  - Cray Research was world leader for many years, Japan also strong in traditional vector machines
  - Cray Research was bought by Tera Corporation in late 90's
- Modern 'supercomputers' are now multiprocessors where each processor has one or more 'vector units'
  - Instruction stream divided into scalar operations and vector operations
  - Vector operations executed by vector units – multiple vector units can execute multiple vector operations in parallel
- SIMD support is now mainstream
  - MMX, SSE, SSE2 instructions in X86 are SIMD instructions

MMX Instructions: SIMD Integer Operations

Added eight 64 bit registers. The 64 bit register can be viewed as containing 8 packed bytes, 4 packed words, 2 dwords, or 1 quad.

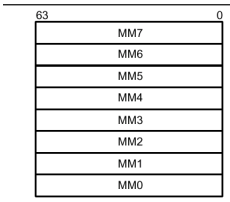


Figure 8-1. MMX™ Register Set

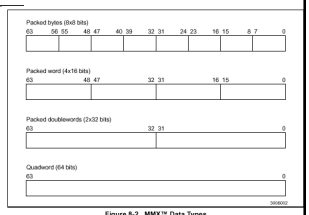


Figure 8-2. MMX™ Data Types

First appeared on Pentium I

SSE Instructions for P3: SIMD Floating Point

New 128 bit registers are called XMM registers (XMM0 – XMM7)
Holds four 32-bit single precision floating point numbers

An instruction like ADDPS xmm0, xmm1 will add the two registers together, computing the sums of the four numbers.

Easy to see speed advantage over previous instructions

	4.0 (32 bits)	4.0 (32 bits)	3.5 (32 bits)	-2.0 (32 bits)
+	-1.5 (32 bits)	2.0 (32 bits)	1.7 (32 bits)	2.3 (32 bits)
	2.5 (32 bits)	6.0 (32 bits)	5.2 (32 bits)	0.3 (32 bits)

SSE P3 SIMD Extensions

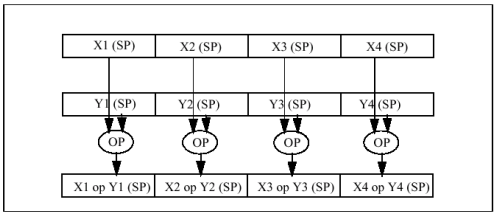


Figure 9-5. Packed Operations

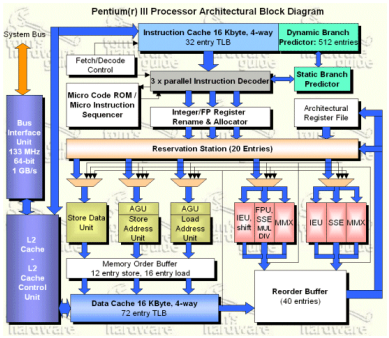
More than 70 instructions. Arithmetic Operations supported: Addition, Subtraction, Mult, Division, Square Root, Maximum, Minimum. Can operate on Floating point or Integer data.

Superscalar vs. VLIW

- The goal of both Superscalar and VLIW architectures is to execute more than one instruction per clock
  - VLIW: Very Long Instruction Word
- Superscalar architectures allow any sequence of instructions
  - Control logic *dynamically schedules* code stream on execution units, resolves all hazards
  - Control is very complicated
- VLIW architectures have simpler control, principally rely on static scheduling
  - Possible to write code sequences that produce incorrect results
  - Compiler must generate code that is hazard free
  - Performance directly tied to quality of compiler generated code

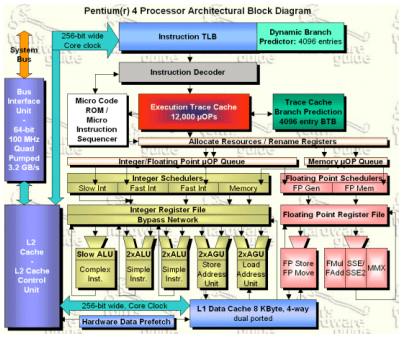
Why Superscalar? Why VLIW?

- Superscalar approaches are used when trying to support an older ISA with a large body of legacy code
  - X86 ISA is a good example – Pentiums were first superscalar implementations of the X86 ISA
  - Have to support legacy applications compiled back in early 80's!
- VLIW approaches preferred for new ISAs
  - Designers can choose which hazards to resolve via compiler, which hazards to resolve in hardware
- Legacy code streams from older ISAs can be handled by a VLIW architecture
  - One approach is dynamic code interpretation/translation to VLIW code stream (IA-64, Pentium 4, Transmeta CPUs and even the Pentium 3 all use this approach)



Pentium III

From www.tomshardware.com



From www.tomshardware.com

Pentium IV

Native instruction set called IA-32.

Pentium 3 vs. Pentium 4 Differences

- L1 cache
  - P3: 16KB instruction, 16KB data
  - P4: 8KB instruction, 16KB data
- P3: X86 instructions translated to pipeline micro-operations dynamically by decoder logic
  - translations are not cached, done every time X86 instruction is executed
- P4: X86 instructions also dynamically translated to micro-operations
  - translation cache holds translations so that X86 instructions do not have to be repeatedly translated
  - should speed execution of tight loops

Pentium 3 vs. Pentium 4 Differences

- P3: 10 pipeline stages, P4: 20 pipeline stages
  - Finer grain pipeline means higher clock speed
  - P4 can have up to 126 instructions “in-flight” including up to 48 load and 24 store instructions
  - Pipeline flushes expensive!!!!
- SIMD Floating point instructions improved quite a bit from P3 (SSE) to P4 (SSE2)
- SSE 128 bit registers can be viewed as these data types
  - 4 single precision FP values (SSE)
  - 2 double precision FP values (SSE2)
  - 16 byte values (SSE2)
  - 8 word values (SSE2)
  - 4 double word values (SSE2)
  - 1 128-bit integer value (SSE2)

Pentium 4 Performance

- For X86 Legacy code, published benchmarks show that a P4 @ 1.4 Ghz is about the same as a P3 @ 1.1 Ghz
  - Somewhat puzzling since the execution trace cache of the P4 should help with execution of X86 instructions
  - Other factors must be involved – L1 cache size? Pipeline flush cost for X86 instructions? Micro Operation structure?
- P4 clock speeds projected to ramp up to 2.0 Ghz by 3rd quarter 2001
- Applications optimized for SSE2 instructions in P4 are 1.5X to 2X faster than P3 (especially double precision floating point)

Transmeta Corp 'Crusoe' Processor

- VLIW Processor for executing x86 code
- Uses a dynamic code translation mechanism to translate segments of X86 code into VLIW code
- Translation cache allows translation to be done once
- Claim is that Crusoe architecture is simpler than Superscalar X86 implementation
  - Less transistors, smaller die, cheaper die
  - Less power
  - Also has additional features for power consumption
- Aimed at mobile computing

Cruosoe Execution Units

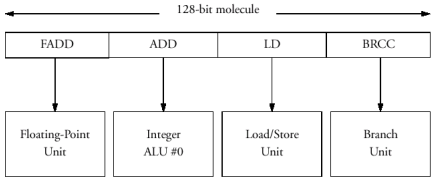


Figure 1. A molecule can contain up to four atoms, which are executed in parallel.

A 'molecule' is a VLIW instruction. Has 64 registers.

Die Sizes

	Mobile PII	Mobile PII	Mobile PIII	TM3120	TM5400
Process	.25m	.25m shrink	.18m	.22m	.18m
On-chip L1 Cache	32KB	32KB	32KB	96KB	128KB
On-chip L2 Cache	0	256KB	256KB	0	256KB
Die Size	130mm^2	180mm^2	106mm^2	77mm^2	73mm^2

Table 1. The Code Morphing software simplifies chip hardware.

Smaller Dies for Crusoe processors, more on-chip cache.

Die Temperature

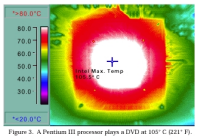


Figure 3. A Pentium III processor plays a DVD at 105° C (221° F).

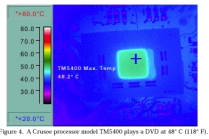


Figure 4. A Crusoe processor model TM5400 plays a DVD at 48° C (118° F).

Cruosoe uses less power.

Code Translation Mechanism

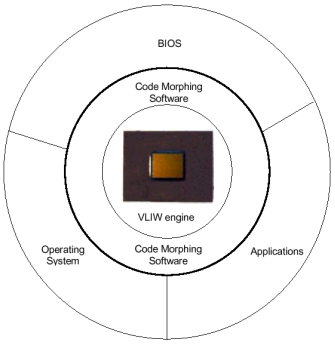


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Comments on 'Code Morphing'

- Code morphing is just translation of X86 code to native Crusoe code
- Each version of the Crusoe chip needs a different version of the translation code
- Crusoe chip is essentially running a very efficient X86 emulator. The design of the Crusoe chip was made with emulation as the end goal, so emulation is very efficient.
  - X86 emulators exist for other architectures (i.e, Macintosh PowerPCs, but as an afterthought and they are inefficient).

Translation Cache

- The code morphing software (emulator) dynamically translates X86 code to Crusoe Code
- Uses a translation cache to exploit the fact that once a loop has been translated, will be executed many times so translation overhead is only once, during first execution of loop
  - Some code (a little) has either large loops or only a few loops and so overhead of translation mechanism impacts performance

Translation

```
A. addl %eax,(%esp)          // load data from stack, add to %eax
B. addl %ebx,(%esp)          // ditto, for %ebx
C. movl %esi,(%ebp)          // load %esi from memory
D. subl %ecx,5               // subtract 5 from %ecx register
```

```
ld %r30,[%esp]               // load from stack, into temporary
add.c %eax,%eax,%r30         // add to %eax, set condition codes.
ld %r31,[%esp]
add.c %ebx,%ebx,%r31
ld %esi,[%ebp]
sub.c %ecx,%ecx,5
```

First pass is simple translation

Translation (continued)

```
ld %r30,[%esp]                // load from stack, into temporary
add.c %eax,%eax,%r30          // add to %eax, set condition codes.
ld %r31,[%esp]
add.c %ebx,%ebx,%r31
ld %esi,[%ebp]
sub.c %ecx,%ecx,5
```

2nd pass, optimizations. Common sub-expression elimination (read stack only once). Move operations when condition codes not needed (only last CC actually needed).

```
ld %r30,[%esp]                // load from stack only once
add %eax,%eax,%r30
add %ebx,%ebx,%r30            // reuse data loaded earlier
ld %esi,[%ebp]
sub.c %ecx,%ecx,5             // only this last condition code needed
```


Translation (continued)

```
ld %r30,[%esp]                // load from stack only once
add %eax,%eax,%r30
add %ebx,%ebx,%r30            // reuse data loaded earlier
ld %esi,[%ebp]
sub.c %ecx,%ecx,5             // only this last condition code needed
```

Group into VLIW instructions for scheduling.

```
1. ld %r30,[%esp];    sub.c %ecx,%ecx,5
2. ld %esi,[%ebp];    add %eax,%eax,%r30;    add %ebx,%ebx,%r30
```


Shadow Registers, Commit/Rollback

Exceptions (i.e. divide by zero) can cause problems since code is aggressively restructured and operations occur out of order.

- Architecture has a set of *shadow* registers. Before executing a block of code in which an exception can occur, copy machine state to exception registers and aggressively translate/execute code
- If code executes with no exception, 'commit' the code block and copy current registers to shadow registers.

Shadow Registers, Commit/Rollback

- If exception occurs, use Shadow registers to restore processor state to previous stable state
- Translate code to version that can handle exception when it occurs, re-execute code
- To handle load/stores, there is also a 'Shadow' memory that handles load/store operations in a similar manner
- Commit/Rollback with Shadow Registers


```
 1.           movl    %ecx,$0x3
 2.           jmp     lbl1
  ...
 3.   lbl1:   movl    %edx,0x2fc(%ebp)
 4.           movl    %eax,0x304(%ebp)
 5.           movl    %esi,$0x0
 6.           cmpl    %edx,%eax
 7.           movl    0x40(%esp,1),$0x0
 8.           jle     skip1
 9.           movl    %esi,$0x1
10.   skip1: movl    0x6c(%esp,1),%esi
11.           cmpl    %edx,%eax
12.           movl    %eax,$0x1
13.           jl      skip2
14.           xorl    %eax,%eax
15.   skip2: movl    %esi,0x308(%ebp)
16.           movl    %edi,0x300(%ebp)
17.           movl    0x7c(%esp,1),%eax
18.           cmpl    %esi,%edi
19.           movl    %eax,$0x0
20.           jnl     exit1
        exit2:
```

Another Example
Comments on translation

a. No jmp
b. Reg. Renamed
c. Out-of-order execution
d.Both code branches executed. Select instruction used to choose result.

```
 1.   addi   %r39,%ebp,0x2fc
 2.   addi   %r38,%ebp,0x304
 3.   ld     %edx,[%r39];      add %r27,%r38,4;     add %r26,%r38,-4
 4.   ld     %r31,[%r38];      add %r35,0,1;        add %r36,%esp,0x40
 5.   ldp    %esi,[%r27];      add %r33,%esp,0x6c;  sub.c %null,%edx,%r31
 6.   ldp    %edi,[%r26];      sel %ile,%r32,0,%r35
 7.   stam   0,[%r36];         sel #1,%r24,%r35,0;  add %r25,%esp,0x7c
 8.   stam   %r32,[%r33];      add %ecx,0,3;        sub.c %null,%esi,%edi
 9.   st     %r24,[%r25];      or %eax,0,0;         brcc #lt,<exit2>
10.   br  <exit1>
```

6

Additional Power Savings

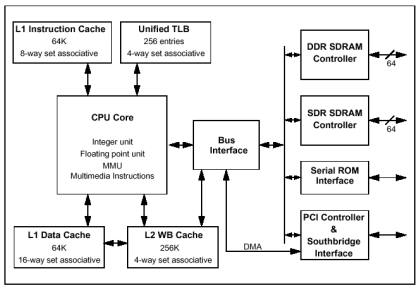
- Translation software (Code morpher) is actually a psuedo operating system
- Code morphing dynamically adjusts both clock frequency AND power supply to adjust to execution speed needs
  - power varies linearly with clock frequency
  - power varies with square of voltage
  - lowering both frequency and power results in cubic power savings
- Most other processors only vary clock frequency.
- Claim power savings on typical applications of 30% (other processors have 10% savings).

Adjusting power to meet user demands

- How do you know if you are running code fast enough?
- Real time software (i.e. DVD player) is easy – just run it fast enough to keep up with streaming data
- User interface software – a bit more heuristic, store profiles to allow user to help specify if performance 'is good enough'.

TMS5400

FIGURE 1. Crusoe Processor Block Diagram - Model TM5400



Integer pipeline is 7 stage, FP pipeline is 10 stage.

TMS5400 vs TMS 3200

	TM3200	TM5400
Frequency Range	333-400MHz	500-700MHz
L1 Cache	96KB	128KB
L2 Cache		256KB
Main Memory	SDRAM (66 to 133MHz)	DDR-SDRAM (100 to 166MHz)
Upgrade Memory		SDRAM (66 to 133MHz)
North Bridge	Integrated	Integrated
Package	474 BGA	474 BGA
Sample	Now	Now
Production	Now	Now