UT 1. Introduction to Computer Architecture

Lecture 1.3 Instruction set architecture design

J. Duato, J. Flich, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Department of Computer Engineering
Universitat Politècnica de València

Contents

1. General aspects related to instruction sets
2. Types of instruction sets
3. Registers and Operand types
4. Instruction encoding
5. Memory addressing
6. Operands and operations
7. Control flow
8. MIPS64 instruction set
9. SIMD Instructions

Bibliography (1/2)


Bostjan Cigan.
SIMD SSE instructions in C, part one, April 2012.


John L. Hennessy and David A. Patterson.
Computer Architecture, Fourth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4 edition, 2006.


John L. Hennessy and David A. Patterson.
Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.


Microsoft.
MMX, SSE, and SSE2 intrinsics, April 2013.

Bibliography (2/2)

Microsoft.
Streaming SIMD extensions 4 instructions, April 2013.

Microsoft.
Supplemental streaming SIMD extensions 3 instructions, April 2013.

Contents

1 General aspects related to instruction sets

2 Types of instruction sets

3 Registers and Operand types

4 Instruction encoding

5 Memory addressing

6 Operands and operations

7 Control flow

8 MIPS64 instruction set

9 SIMD Instructions

Instruction set design factors

- Instruction sets act as interfaces between programs and datapaths of processors
- Instructions results from program compilation
  $\rightarrow$ Those instructions rarely used by compilers are useless
- Experience in program compilation shows that
  $\rightarrow$ Although programs can be very complex, most of them are very simple
- *F*ast execution of simple instructions possible using simple datapaths $\rightarrow$ CPI, T $\downarrow$

Main features of instruction sets (1/2)

Basic principle

"Common case: efficient; Uncommon case: correct"

Ortogonality

Whenever it makes sense, operations, addressing modes and data types must be independent.

→ Simplify generation of code, in particular when decisions involve two different compilation phases

Provision of primitives instead of solutions

Avoid the inclusion of solutions directly supporting high-level instructions

→ Attending to their specificity, such solutions only work for particular programming languages

→ They provide more or less functionality than necessary

Instead, instruction sets must provide to compilers primitives rather to solutions for the generation of optimized code

Main features of instruction sets (2/2)

Principle "one or all"

Either there is only one way to make something, or all ways are possible.

→ Reduce the cost of computing each alternative

Example: branch conditions

- $>, =$ One way to code branch conditions
- $>, >=, <, <=, =, <>$ All possible ways to code branch conditions

Integrate instructions operating on constants for those values already known at compile-time

→ Computation of such values at runtime is not efficient.

Contents

1 General aspects related to instruction sets

2 Types of instruction sets

3 Registers and Operand types

4 Instruction encoding

5 Memory addressing

6 Operands and operations

7 Control flow

8 MIPS64 instruction set

9 SIMD Instructions

Basic Paradigms (1/2)

Classic criteria: storage of operands in the CPU.
Instructions compute on input data and produce a result.
Input data and results can be stored in memory and *CPU*.

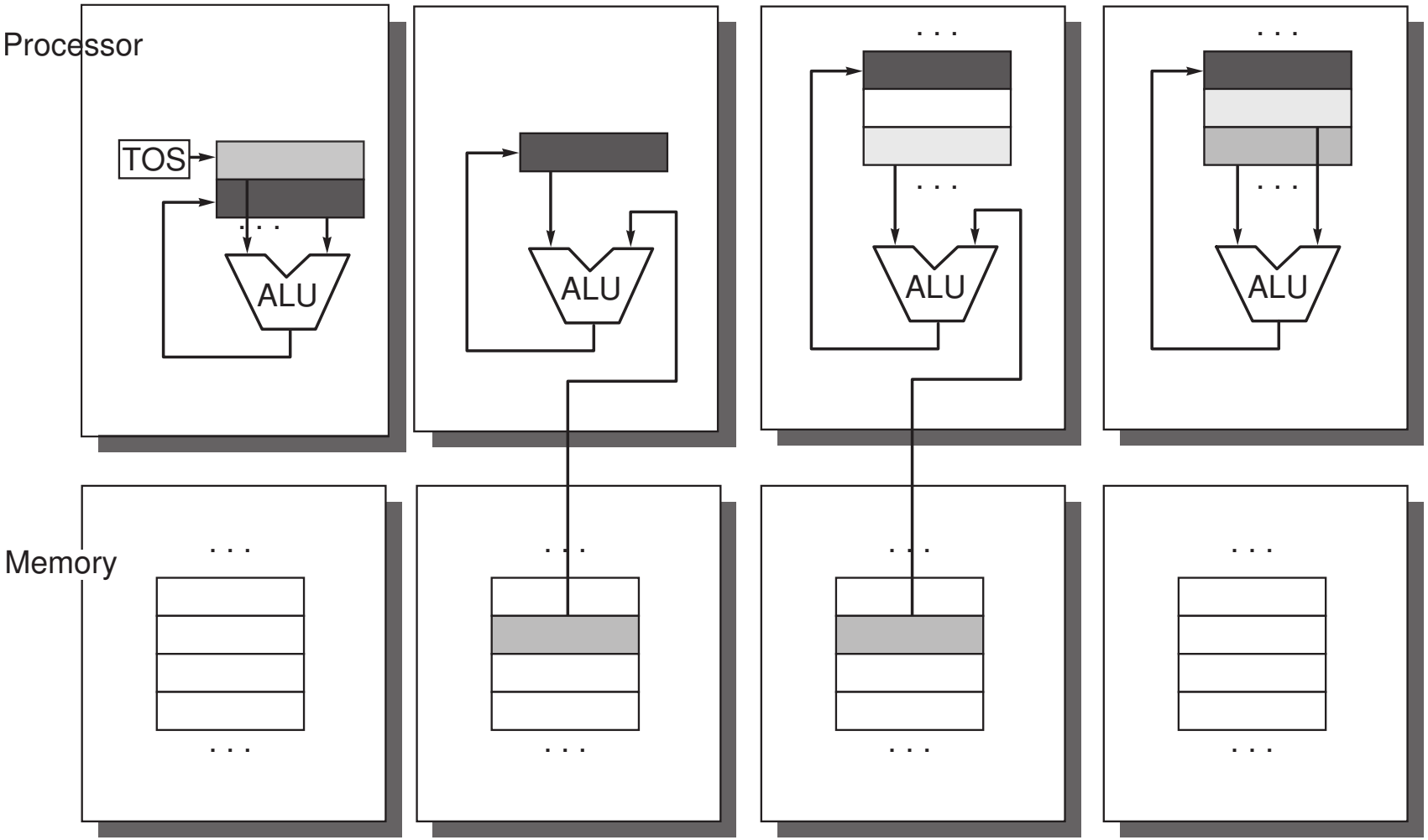
Three paradigms:

Stack (Ex: HP 3000)
Input data and results are stored in the stack. (Implicit) operands are stored in the stack head and replaced by the result.

Accumulator (Ex: PDP-8, M6809).
There is an (implicit) *accumulator* register storing one of the operands and the final result.

General purpose registers (Ex: PDP-11).
Operands and results are in a *register file*. All operands must be explicitly referenced.

Basic Paradigms (2/2)



(a) Stack (b) Accumulator (c) Register-memory (d) Register-register/ load-store

Basic Paradigms

Code for $C = A + B$

Stack	Accumulator	General purpose registers		
`PUSH A`	`LOAD A`	`LOAD R1,A`	`LOAD R1,A`	`MOVE C,A`
`PUSH B`	`ADD B`	`ADD R1,B`	`LOAD R2,B`	`ADD C,B`
`ADD`	`STORE C`	`STORE C,R1`	`ADD R3,R1,R2`	
`POP C`			`STORE C,R3`	

- Current instruction sets use a general purpose register file
- Other paradigms (Stack, accumulator) has disappeared

$\Rightarrow$ The paradigm of processor with general purpose register file and addressable memory allows an efficient compilation of programs

Computers using general purpose registers: a taxonomy (1/2)

Two relevant parameters:

1. n operands in ALU instructions (2 or 3).
   - If $n = 2$, then one operand acts as source and destination
   - If $n = 3$, then there are two sources and one destination.
2. Number $m \leq n$ of memory addresses in ALU instructions

Cases:

- If $m = 0$, *load/store reg–reg* computers (typically $m = 0, n = 3$).
  - All instructions work with register data and store results in registers.
  - Only *load* and *store* instructions exchange information with memory.
- If $m < n$, *reg–mem* computers (typically $m = 1, n = 2$).
- If $m = n$, *mem–mem* computers (typically $m = 3, n = 3$).

Computers using general purpose registers: a taxonomy (2/2)

The choice impacts the execution time $T_{ex} = I \times CPI \times T$

Two main types of instructions sets:

IA (*Intel Architecture*) Intel and compatible (AMD) processors

RISC Used in MIPS, ARM, SPARC, POWER (already commercial) and HP-PA, Alpha (extinguished) processors

Main features:

	Intel Architecture	**RISC**
Prog. model	*R-M* (Register-memory)	*L/S* (Load/Store)
Registers	Few, variable length	A lot, fixed length
Instr. format	Variable	Fixed (32 bits)

L/S and R-M models (1/3)

RISC **L/S Model**

1. ALU instructions compute only on registers
2. ALU instructions include three operands
3. *Load* and *Store* instructions exchange data between registers and memory
4. The amount of work performed by all instructions is similar (a computation or a memory access)

IA **R-M Model**

1. Instructions work on registers, or with an operand in memory
2. Instructions have two operands
3. The MOV instruction exchange data $R \Leftrightarrow M$ and $R \Leftrightarrow R$
4. The amount of work performed by instructions is quite diverse

L/S and R-M models (2/3)

How do these model affect the execution time?

RISC **L/S Model**

1. A program has more instructions to carry out the same work $\rightarrow I \uparrow$
2. Simpler format, easier decoding $\rightarrow T \downarrow$
3. The amount of work per instruction remains similar $\rightarrow CPI \downarrow$

IA **R-M Model**

1. A program integrates less instructions to perform the same work $\rightarrow I \downarrow$
2. Variable format, more complex decoding $\rightarrow T \uparrow$
3. Heterogeneous amount of work per instructions $\rightarrow CPI \uparrow$

L/S and R-M models (3/3)

Example

Intel 32 bits code	Equivalent MIPS-32 code
`add EAX,EBX`	`add $t0,$t0,$t1`
`add EAX,a`	`lw $t1,a` `add $t0,$t0,$t1`

`add a,EAX`	`lw $t1,a`
	`add $t1,$t1,$t0`
	`sw $t1,a`

Contents

1 General aspects related to instruction sets

2 Types of instruction sets

3 Registers and Operand types

4 Instruction encoding

5 Memory addressing

6 Operands and operations

7 Control flow

8 MIPS64 instruction set

9 SIMD Instructions

Registers and Operand types (1/4)

RISC Big number of registers, all of them with the same length.
Example: MIPS r2000: 32 integer registers (32 bits) and 32 floating point registers (32 bits)
Regular instructions use the whole content of registers.
L/S instructions transform integer types whenever necessary

IA Few registers that are adapted to the available data types
Each ALU instruction has a different operation code for each data type
In x86-64 there are 4 versions of `add` for operands of 8, 16, 32 and 64 bits

$\rightarrow$ Changing the word length is simpler in RISC computers

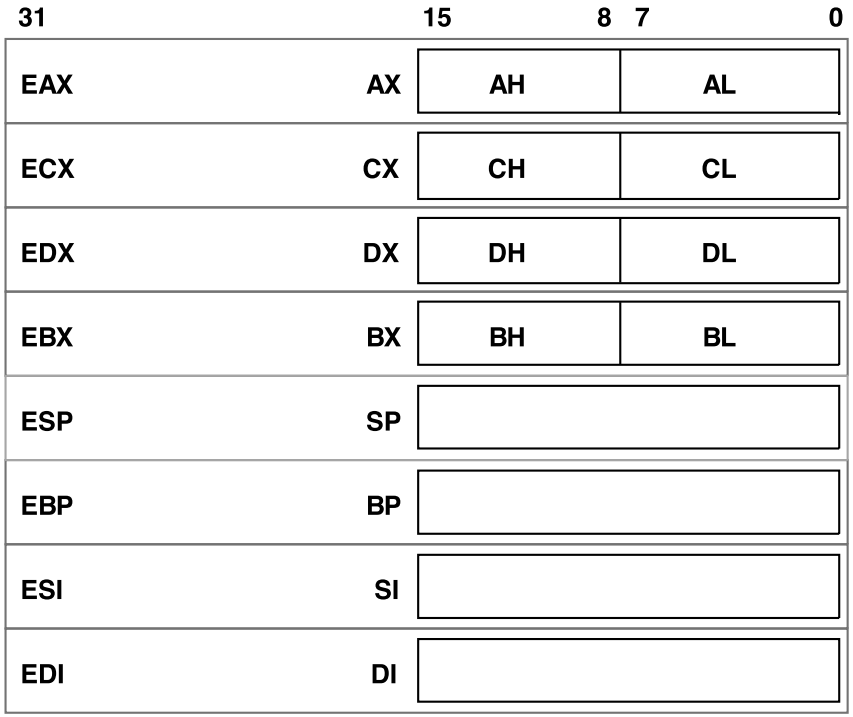
Registers and Operand types (2/4)

Example: IA-32 registers

- 8 registers of 32 bits EAX ... EDI
- Low part of 4 of them can be managed as 4 registers of 16 bits or 8 registers of 8 bits

Extension to 64 bits:

- 8 registers of 32 bits EAX ... EDI are the low part of existing 8 registers of 64 bits RAX ... RDI
- There are 8 additional registers of 64 bits



Registers and Operand types (3/4)

Example: MIPS registers

- 32 register of 32 bits: R0 ... R31
- Agreement typically used by compilers

Name	Number	Use	Preserved across a call?
$zero	0	The constant value 0	N.A.
$at	1	Assembler temporary	No
$v0–$v1	2–3	Values for function results and expression evaluation	No
$a0–$a3	4–7	Arguments	No
$t0–$t7	8–15	Temporaries	No
$s0–$s7	16–23	Saved temporaries	Yes
$t8–$t9	24–25	Temporaries	No
$k0–$k1	26–27	Reserved for OS kernel	No
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	30	Frame pointer	Yes
$ra	31	Return address	Yes

Registers and Operand types (4/4)

Example

Source code

```
byte a,b,c;
...
c = a + b;
```

IA-32

```
a   db ...
    ...
    MOV AL,a
    add AL,b
    MOV c,AL
```

MIPS r2000

```
a:  byte ...
    ...
    lb $t0,a
    lb $t1,b
    add $t2,$t0,$t1
    sb $t2,c
```

Contents

1 General aspects related to instruction sets

2 Types of instruction sets

3 Registers and Operand types

4 Instruction encoding

5 Memory addressing

6 Operands and operations

7 Control flow

8 MIPS64 instruction set

9 SIMD Instructions

Encoding strategies (1/5)

Instructions are stored in memory according to a format indicating the operation to carry out (operation code) and the operands

Fixed vs. variable format

Fixed All instructions are encoded using the same number of bits.

- It easies fetching and decoding of instructions
- Sometimes, it wastes bits in the format, since all instructions do not require the same amount of bits to be encoded.

Variable The required number of bits for encoding varies attending to the instruction type.

- Space taken by instructions (and programs) is optimized.
- It makes more complex fetching and decoding of instructions.

Encoding strategies (2/5)

Number of bits of the format:

The number of bits of the format limits the space devoted to each instruction field, which limits the number of available variants:

- number of instructions (operation codes),
- number of registers,
- addressable memory space,
- etc.

Encoding strategies (3/5)

Number of instruction formats

How are format bits assigned to instruction fields?

One format. The correspondence between format bits and fields is always the same.

- Easies instruction decoding.
- Sometimes, format bits are wasted, since not all instructions require all the available fields.

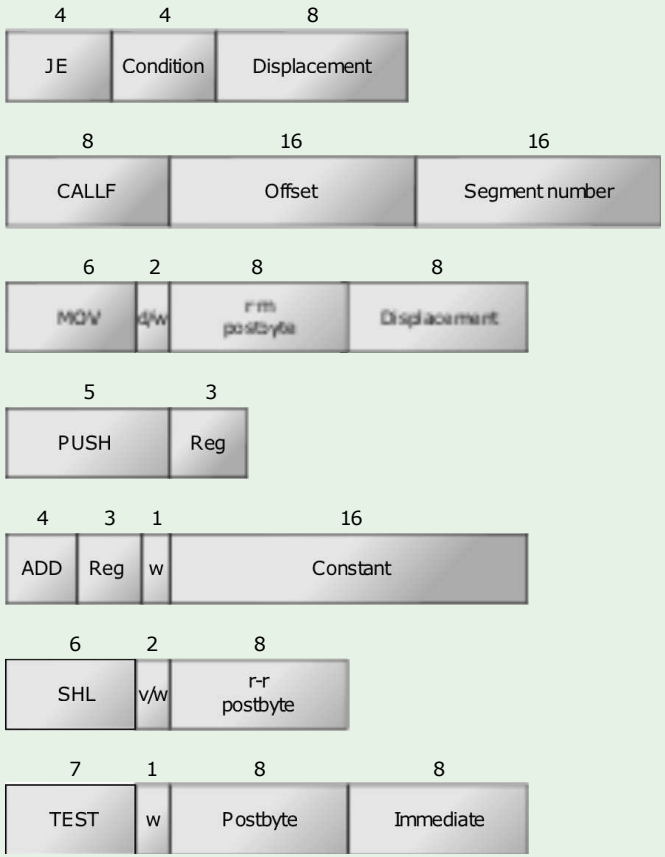
Multiple formats. Each format can have different fields and establishes a correspondence between such fields and the format bits.

- Allow a better adjustment of those bits taken by each instruction and the required fields.

Encoding strategies (4/5)

IA-32 encoding

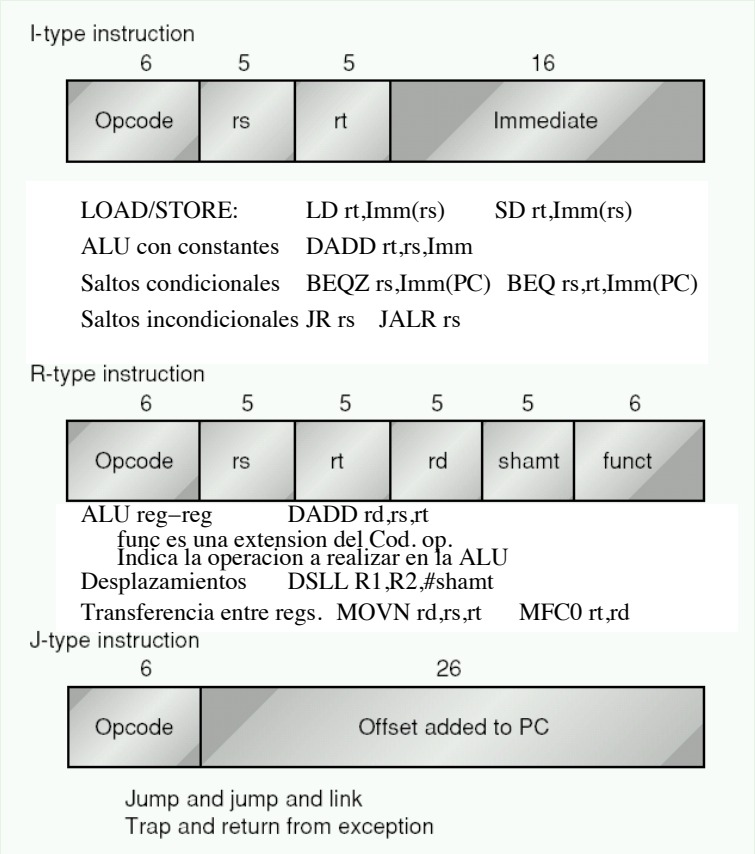
- Variable format: one instructions can take from 1 to 17 bytes
- The decoding process is sequential: the value of bits in one field must be known in order to decode the following one



Encoding strategies (5/5)

MIPS64 encoding

- 32 bits fixed format.
- Fields can be decoded in parallel
- 3 format types: R, I and J
- Number of bits. Up to 2^6 opcode + 2^6 extensions (R format); Up to 2^5 registers; branches and immediate values of 2^{16}.



Contents

1 General aspects related to instruction sets

2 Types of instruction sets

3 Registers and Operand types

4 Instruction encoding

5 Memory addressing

6 Operands and operations

7 Control flow

8 MIPS64 instruction set

9 SIMD Instructions

Address interpretation

- Physical reading/writing units (words) contain $W = 2^w$ addressable units (bytes)
- Words referred through the address of their least significant byte.
  $\rightarrow$ The word with address A contains the bytes with addresses $A, A+1, \ldots, A+W-1$
- Two alternatives, with no important consequence are:

IA **Little endian**

Byte with address A is located at the least significant word position

Word Addresses	Byte Addresses			
0	3	2	1	0
4	7	6	5	4

RISC **Big endian**

Byte with address A is located at the most significant word position

Word Addresses	Byte Addresses			
0	0	1	2	3
4	4	5	6	7

Aligment

- Instruction sets provide access to units of $1, 2, \ldots 2^w$ bytes
  In the MIPS-32: byte, *halfword*, *word*, etc.
- Two alternatives:

  RISC **Aligned access**

  The address of the object of 2^i bytes is multiple of 2^i
  In the MIPS-32: the address of a *halfword* is always even, and the address of a *word* is a multiple of 4

  IA **Non-aligned access**

  There is no restriction
  An instruction can access contiguous byte in two consecutive words, and its execution will demand two physical accesses to memory
  $\rightarrow$ Impact on the HW complexity: $CPI \uparrow$, $T \uparrow$

Addressing modes (1/6)

How are instruction operands specified? $\rightarrow$ addressing modes
To what extend is the support of sophisticated addressing modes interesting?

- It reduces the number of program instructions $\rightarrow$ $I \downarrow$
- More complex *Hardware* $\rightarrow$ $CPI \uparrow$ and/or $T \uparrow$.

Addressing modes (2/6)

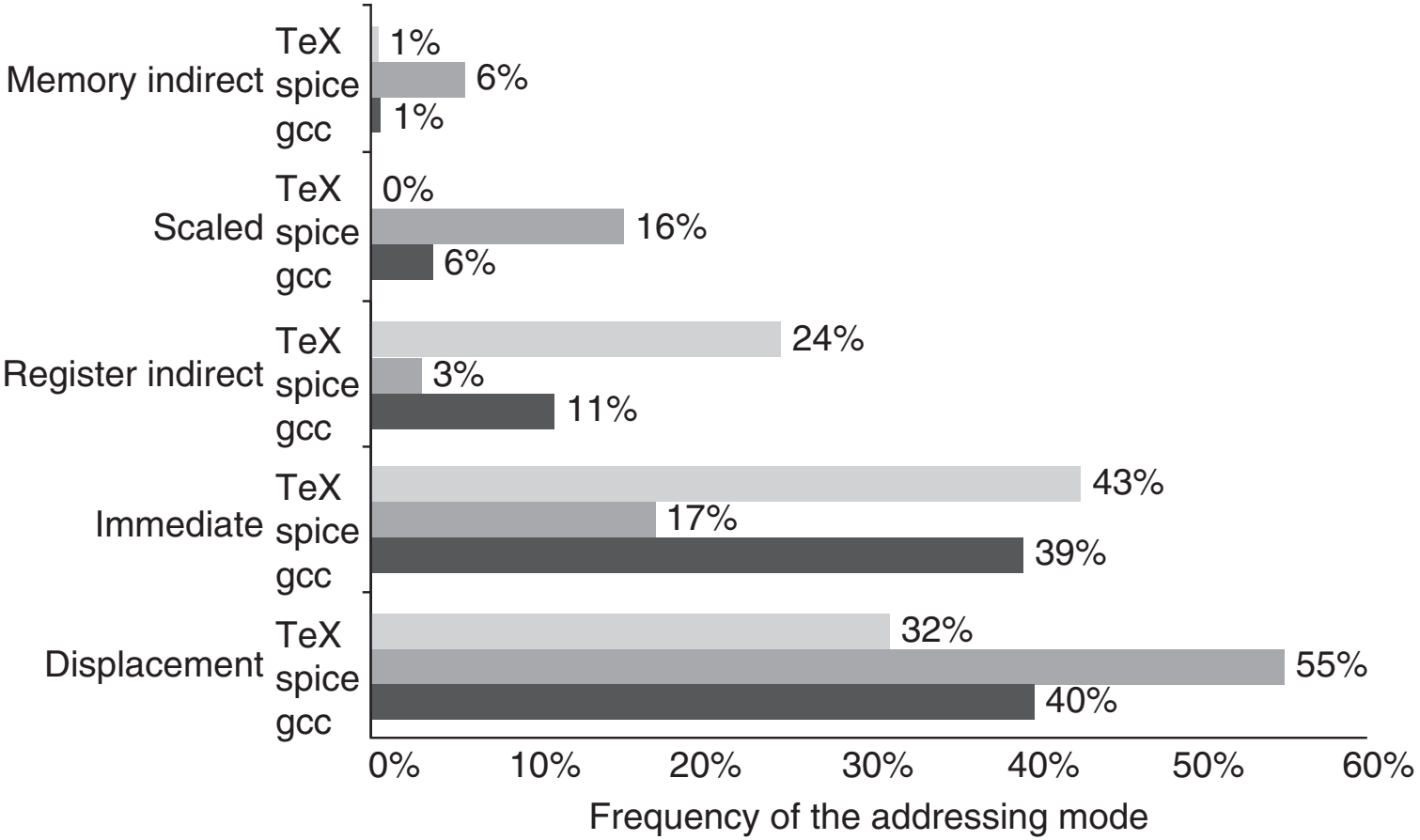
Some examples of addressing modes

Mode	Example	Meaning
Register direct	add r1,r2,r3	r1←r2+r3
Immediate/literal	add r1,r2,#1	r1←r2+1
Absolute/Direct	lw r1,(1000)	r1←Mem[1000]
Register indirect	lw r1,(r2)	r1←Mem[r2]
Indexed absolute/Displacement	lw r1,100(r2)	r1←Mem[100+r2]
Base plus index / Indexed	lw r1,(r2+r3)	r1←Mem[r2+r3]
Memory indirect	lw r1,@(r2)	r1←Mem[Mem[r2]]
Autoincrement register indirect	lw r1,(r2)+	r1←Mem[r2] r2←r2+*d*
Autodecrement register indirect	lw r1,–(r2)	r2←r2–*d* r1←Mem[r2]
Scaled	lw r1,100(r2)(r3)	r1←Mem[100+r2+r3**d*]

d: operand size in bytes

Addressing modes (3/6)

Statistics reflecting the use of addressing modes in programs:



Addressing modes (4/6)

IA x86-64 includes even the scaled mode ([Reg]+[Reg]×d+displacement)
Freedom to combine the addressing modes with operation codes.
With the most complex modes, an instruction may have a lot of work:

Example:

Intel 32 bits	**Equivalent MIPS-32 code**

`add EAX,[BX][SI].X`	`add $t0,$t0,$t1`
	`lw $t2,X($t0)`
	`add $t3,$t3,$t2`

`add EAX,a`	`lw $t1,a` `add $t0,$t0,$t1`

Autoincrement and autodecrement register indirect modes are not available

Addressing modes (5/6)

RISC Simplifying:

- Immediate mode: two versions of ALU instructions available:

  R-format : Rd $\Leftarrow$ Rs op Rt.

  Example: `ADD`

  I-format : Rd $\Leftarrow$ Rs op X

  Example: `ADDI`

  Range of available values: a solution that must balance the number of bits taken and the magnitude of the constants required. 16 bits in the MIPS.

  However, solutions are also provided to easy the work with bigger immediate values. For instance:
  `LUI R1,#Uvalue`: Regs[R1]$\leftarrow$ value $<<$ 16 `ORI R1,R1,#Lvalue`: Regs[R1]$\leftarrow$ Regs[R1] **or** Lvalue

Addressing modes (6/6)

- Main memory accesses must be performed using the displacement addressing mode: `X(Rn)`
  - Using X=0 results in the register indirect addressing mode
  - Using Rn = $zero results in the absolute mode

  Displacement range: solution that must balance the size of the necessary displacements and the number of required bits. 16 bits in the MIPS.

Some RISC processors have indexed addressing modes

Contents

1 General aspects related to instruction sets

2 Types of instruction sets

3 Registers and Operand types

4 Instruction encoding

5 Memory addressing

6 Operands and operations

7 Control flow

8 MIPS64 instruction set

9 SIMD Instructions

Type of operands

- How to code the type of an operand? Normally it is included in the operation code, although some old computers provided operands and labels representing their types.
- Most commonly supported types are: char (8 bits – ASCII), integer (8-*byte*, 16-*half-word*, 32-*word* y 64 bits-*double word*, all of them in 2's complement), floating point (single-precision- 32 bits y double precision -64 bits, in the IEEE 754 standard).
- Occasionally supported types are: charstrips, BCD (*binary coded decimal*), quad-precision floating point (128 bits). For multimedia programs: *vertex* (3 D+color), *pixels* (R+G+B+A). DSP's use to support fixed arithmetic.

A variety of operations

- Integer arithmetic, binary logic.
- Transfer: *Load/store* or *move*.
- Control: branches and jumps, call/return.
- System: OS calls, virtual memory management.
- Floating point: arithmetic and real-integer conversions.
- Decimal: arithmetic and decimal-char conversions
- Strings: transference, comparison, search.
- Multimedia: *vertex y pixels* operations, compression/decompression, SIMD instructions.

Contents

1 General aspects related to instruction sets

2 Types of instruction sets

3 Registers and Operand types

4 Instruction encoding

5 Memory addressing

6 Operands and operations

7 Control flow

8 MIPS64 instruction set

9 SIMD Instructions

Control instructions

Types:

- Conditional branches (*branch*),
- Unconditional branches (*jump*),
- *Call/return* to/from a procedure

Statistics of use:

- *jump*, *call* and *return* represent $\frac{1}{3}$, and are always taken.
- Conditional branches. Other $\frac{1}{3}$ corresponds to loops, which are nearly 100% of the times taken. The remaining $\frac{1}{3}$ of the branches are taken in 50 % of the cases.

$\rightarrow$ The most likely alternative for a branch is to take it: $\frac{5}{6}$ are taken while only $\frac{1}{6}$ are not.

Addressing modes (1/2)

PC-relative

- Destination use to be near the current instruction $\rightarrow$ relative addresses take few bits.
- Number of bits for encoding the (relative) displacement. Typical values are 16–20 bits in conditional branches and 26 bits in unconditional ones.

Indirect link

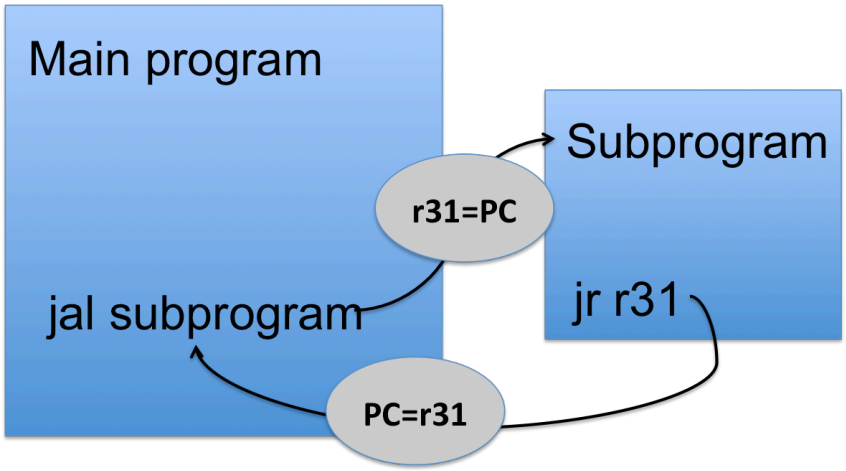
Useful if the branch destination is unknown at compilation time.

- Statements selecting one over several alternatives(i.e., *case. switch*).
- Virtual methods in oo-languages
- Exchange of functions as parameters for other functions
- Dynamically linked libraries.

Addressing modes (2/2)

Jump and Link

- Used to invoke subprograms (*call*).
- It is a branch using a Relative-PC or indirect link addressing mode and storing the return address in a register.
- The subprogram *return* is performed using an indirect link addressing mode. The address (link) stored by the jump instruction in the corresponding register is now used to return.
- Example (MIPS):



Branch conditions (1/2)

How to specify branch **conditions** ?
Several alternatives:

1. Conditions codes (80x86, PowerPC, SPARC).
   - The instruction set defines an “state” that is modified according to the state of the last ALU operation.
   - Tipically there are some condition codes or *flags*: *C* (carry), *Z* (zero), *N* (negative), etc.
   - Branch instructions simply check condition codes.
   - Example (Intel): Branch to `ETI` if $EAX \leq EBX$

```
CMP EAX, EBX # Sub without writing a result
# if EAX <= EBX, then Z=1 or N=1
JLE ETI    # Branch if Z=1 or N=1
```

Branch conditions (2/2)

- Drawbacks:
  - → Generation of condition codes is not trivial and requires space inside the chip
  - → The modification of condition codes by all instructions poses some problems for code reordering and for multiple issuing of ALU instructions

2. Explicit check (MIPS). Result of operations is explicitly checked using specific instructions. Conditions codes do not exist.
   - Examples (MIPS):

```
; comparison, branch
dadd r1, r2, #1
slt r10, r1, r3
bnez r10, dest
```

```
; comparison + branch
dadd r1,r2,#1
bne r1,r3,dest
```

Conditional instructions

Consist of adding an additional operand, the condition, to conventional instructions.

- If the condition is TRUE, the instruction executes normally
- If the condition is FALSE, the instruction behaves as a NOP

Usefulness

Removes branch instructions $\rightarrow$ Reduces I for simple flow control structures.

Examples of conditional instructions

- Conditional transfer **`MOVZ,MOVN`** `Rdst,Rsrc,Rcond`
  $\rightarrow$ If (Rcond=0) then Rdst $\leftarrow$ Rsrc
- Conditional load **`LDC`** `Rdst,disp(Rsrc),Rcond`
  $\rightarrow$ If Rcond then Rdst = Mem[Rsrc+disp]

Use examples

Source code	Conventional code	With cond. instructions

`if (A=0) then`
`  s:= t;`
	`  BNEZ R1, L`
	`  DADD R2, R3, R0`
	`L:`
		`MOVZ R2, R3, R1`

`if (s<0) then`
`  d := -s`
`else`
`  d := s`
	`; R3 = s, R2 = d`
	`DSLT R1,R3,R0`
	`DSUB R2,R0,R3`
	`BNEZ R1,L`
	`DADD R2,R3,R0`
	`L:`
		`; R3 = s, R2 = d`
		`DSLT R1,R3,R0`
		`DSUB R2,R0,R3`
		`MOVZ R2,R3,R1`

	`LD R1,40(R2)`
	`BEQZ R1, L`
	`LD R8,20(R10)`
	`LD R9,0(R8)`
	`L:`
		`LD R1,40(R2)`
		`LDC R8,20(R10),R1`
		`LDC R9,0(R8),R1`
		`L:`

Limitations of conditional instructions

- They spend clock cycles, and therefore, they contribute to I, even if the condition is not met → An excessive use of conditional instructions may end up increasing I.
  Example: If a branch is implemented with conditional instructions, both instruction paths must be executed, while in the conventional code only one instruction path is executed.
- If the control flow leads to the execution of nested conditional sentences, the use of conditional instructions becomes less effective
  Example: Moving an instruction across two branches → either the instruction includes both conditions, or additional instructions will be required in order to combine the conditions
- CPI may be increased or the clock frequency may be reduced

⇒ Many current architectures (e.g., MIPS64) incorporate simple conditional instructions, such as **`MOVZ`** or **`MOVN`**

Contents

1 General aspects related to instruction sets

2 Types of instruction sets

3 Registers and Operand types

4 Instruction encoding

5 Memory addressing

6 Operands and operations

7 Control flow

8 MIPS64 instruction set

9 SIMD Instructions

Arithmetic/Logical/Floating point

Arithmetic/logical	*Operations on integer or logical data in GPRs; signed arithmetic trap on overflow*
DADD, DADDI, DADDU, DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB, DSUBU	Subtract, signed and unsigned
DMUL, DMULU, DDIV, DDIVU, MADD	Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends
DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV	Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic
SLT, SLTI, SLTU, SLTIU	Set less than, set less than immediate, signed and unsigned

Floating point	*FP operations on DP and SP formats*
ADD.D, ADD.S, ADD.PS	Add DP, SP numbers, and pairs of SP numbers
SUB.D, SUB.S, SUB.PS	Subtract DP, SP numbers, and pairs of SP numbers
MUL.D, MUL.S, MUL.PS	Multiply DP, SP floating point, and pairs of SP numbers
MADD.D, MADD.S, MADD.PS	Multiply-add DP, SP numbers, and pairs of SP numbers
DIV.D, DIV.S, DIV.PS	Divide DP, SP floating point, and pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C.__.D, C.__.S	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

Data transfers and control

Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR*
LB, LBU, SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH, LHU, SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW, LWU, SW	Load word, load word unsigned, store word (to/from integer registers)
LD, SD	Load double word, store double word (to/from integer registers)
L.S, L.D, S.S, S.D	Load SP float, load DP float, store SP float, store DP float
MFC0, MTC0	Copy from/to GPR to/from a special register
MOV.S, MOV.D	Copy one SP or DP FP register to another FP register
MFC1, MTC1	Copy 32 bits to/from FP registers from/to integer registers
Control	*Conditional branches and jumps; PC-relative or through register*
BEQZ, BNEZ	Branch GPRs equal/not equal to zero; 16-bit offset from PC + 4
BEQ, BNE	Branch GPR equal/not equal; 16-bit offset from PC + 4
BC1T, BC1F	Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4
MOVN, MOVZ	Copy GPR to another GPR if third GPR is negative, zero
J, JR	Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
ERET	Return to user code from an exception; restore user mode

Contents

1 General aspects related to instruction sets

2 Types of instruction sets

3 Registers and Operand types

4 Instruction encoding

5 Memory addressing

6 Operands and operations

7 Control flow

8 MIPS64 instruction set

9 SIMD Instructions

Vector processors

Many numerical applications require processing large arrays of data.

- Simulation of physical and chemical processes, image and video processing, design of complex structures, etc.

From 1975 to 2000 (approx), several companies commercialized vector supercomputers, with instructions that operated on vectors.

- A vector instruction operates with vectors of a given maximum size (typically 64 or 256), formed by elements of a certain size (32 or 64 bits).
- The types of data they handle are, almost always, FP in single or double precision.

When the integration scale was large enough, microprocessors extended their ISA with SIMD instructions that can operate on short vectors.

Packed data types (1/2)

SIMD *(Single Instruction - Multiple Data)* instructions operate on registers that contain several packed data.

- They support different data types: 8, 16, 32, 64-bit integers, and single and double precision floating point.
- The number of data contained in a register is

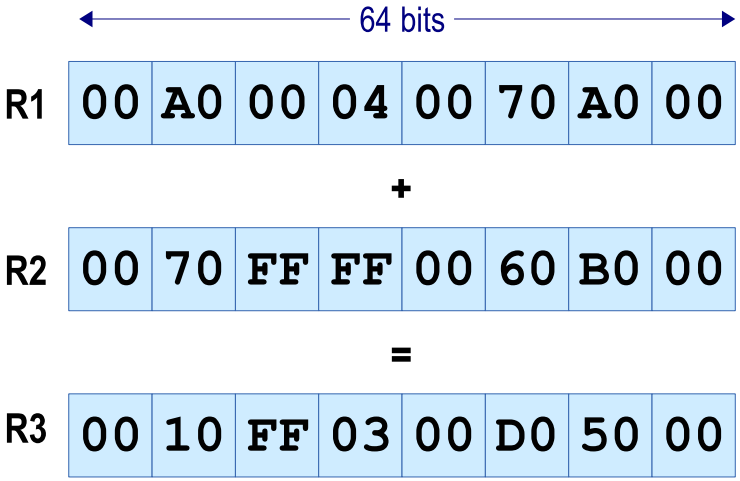
$$n = \frac{\text{register size}}{\text{size of data type}}$$

- The registers have a fixed size, but the instruction set allows packing data of different sizes.
  Example: 64-bit registers can contain
  - 8 data of type *byte*
  - 4 data of type *halfword*
  - 2 data of type *word* or *float*
  - 1 data type *double*

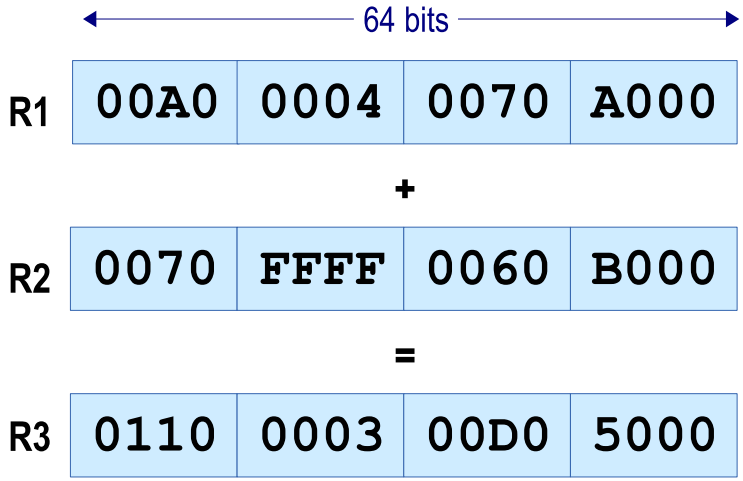
Packed data types (2/2)

A SIMD instruction performs *n* identical operations.
Example of two addition operations with 64-bit registers

`add_pb R3,R1,R2`
(8 data of type byte)



`add_ph R3,R1,R2`
(4 data of type halfword)



Examples of SIMD instructions in several architectures

Instruction category	Alpha MAX	HP PA-RISC MAX2	Intel Pentium MMX	Power PC AltiVec	SPARC VIS
Add/subtract		4H	8B,4H,2W	16B, 8H, 4W	4H,2W
Saturating add/sub		4H	8B,4H	16B, 8H, 4W	
Multiply			4H	16B, 8H	
Compare	8B (>=)		8B,4H,2W (=,>)	16B, 8H, 4W (=,>,>=,<,<=)	4H,2W (=,not=,>,<=)
Shift right/left		4H	4H,2W	16B, 8H, 4W	
Shift right arithmetic		4H		16B, 8H, 4W	
Multiply and add				8H	
Shift and add (saturating)		4H			
And/or/xor	8B,4H,2W	8B,4H,2W	8B,4H,2W	16B, 8H, 4W	8B,4H,2W
Absolute difference	8B			16B, 8H, 4W	8B
Maximum/minimum	8B, 4W			16B, 8H, 4W	
Pack (2n bits --> n bits)	2W->2B, 4H->4B	2*4H->8B	4H->4B, 2W->2H	4W->4B, 8H->8B	2W->2H, 2W->2B, 4H->4B
Unpack/merge	2B->2W, 4B->4H		2B->2W, 4B->4H	4B->4W, 8B->8H	4B->4H, 2*4B->8B
Permute/shuffle		4H		16B, 8H, 4W	

Evolution of SIMD instructions on Intel processors

1997 Pentium MMX with eight 64-bit registers (MM0-MM7).

1999 Pentium-III with SSE instruction extensions. New instructions and 128-bit registers (XMM0-XMM7).

2001 Pentium 4 with SSE2 adds new instructions.

2004 SSE3

2006 SSSE3 and SSE4

2008 AVX, with 256-bit registers renamed as YMM0-YMM15

2011 AVX2

2015 AVX-512, with 32 ZMM registers of size 512 bits.

Trend

A significant percentage of the additional transistors in each new generation of processors is devoted to increasingly powerful vector instructions and larger vector register files.

AVX-512 vector register files

The scheme of the AVX-512 registers (`ZMM`) as an extension of the AVX registers (`YMM`) and the SSE registers (`XMM`) is as follows:

<table>
<tr><td>511 256</td><td>255 128</td><td>127 0</td></tr>
<tr><td colspan="2">ZMM0</td><td>XMM0</td></tr>
<tr><td colspan="2">ZMM1</td><td>XMM1</td></tr>
<tr><td colspan="2">...</td><td>...</td></tr>
<tr><td colspan="2">ZMM15</td><td>XMM15</td></tr>
<tr><td colspan="3">ZMM16</td></tr>
<tr><td colspan="3">...</td></tr>
<tr><td colspan="3">ZMM31</td></tr>
</table>

On processors with support for Intel AVX-512, SSE and AVX instructions operate on the least significant 128 or 256 bit of the first 16 `ZMM` registers.

Example 1: SAXPY

The computation of $\vec{Y} = a\vec{X} + \vec{Y}$ is very common in numerical applications
It is called SAXPY (simple precision) loop or DAXPY (double precision) loop.

```
void SAXPY(int n, float a, float *X, float *Y) {
    int i;
    for(i=0; i<n; i++)
        Y[i] = a*X[i] + Y[i];
}
```

Without vector instructions

n iterations, *n* multiplications and *n* additions.

Example 1: SAXPY with SIMD instructions (1/5)

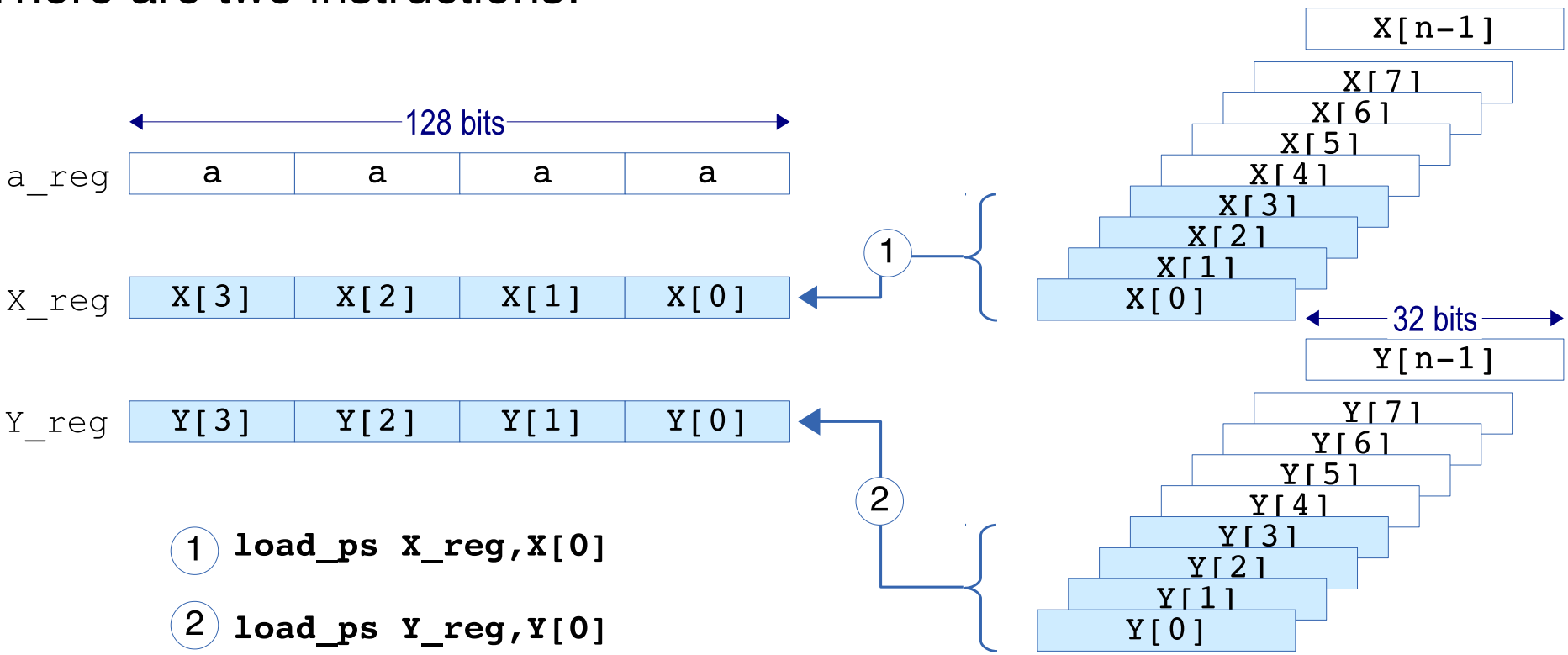
Solution scheme:

- Vector components occupy 32 bits (simple precision)
- 128-bit registers: each one contains a block of four vector components.
- Assume three SIMD registers, named `a_reg`, `X_reg` and `Y_reg`
- The SIMD register `a_reg` must be initialized with four copies of the *a* value
- Consider the instructions:

`load_ps reg,adr`	Reads a block from memory
`add_ps rd,rf1,rf2`	4 component vector addition
`mul_ps rd,rf1,rf2`	4 component vector product
`store_ps reg,adr`	Writes a block into memory

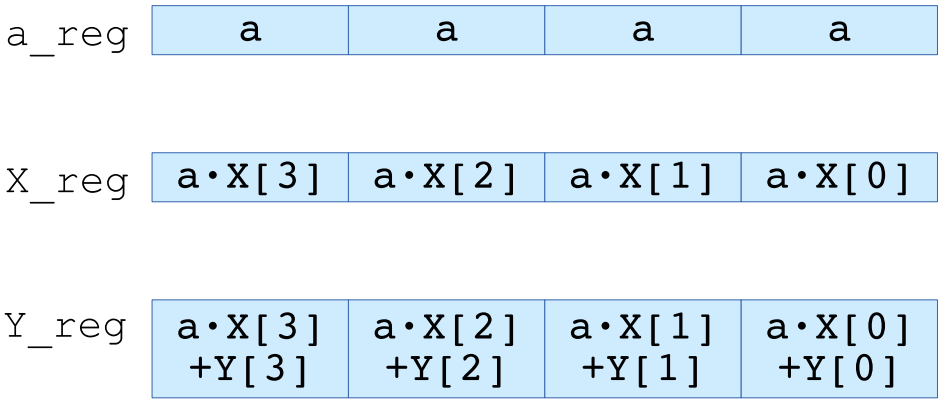
Example 1: SAXPY with SIMD instructions (2/5)

The loop starts by reading 4 components of each vector and storing them in a SIMD register
There are two instructions:



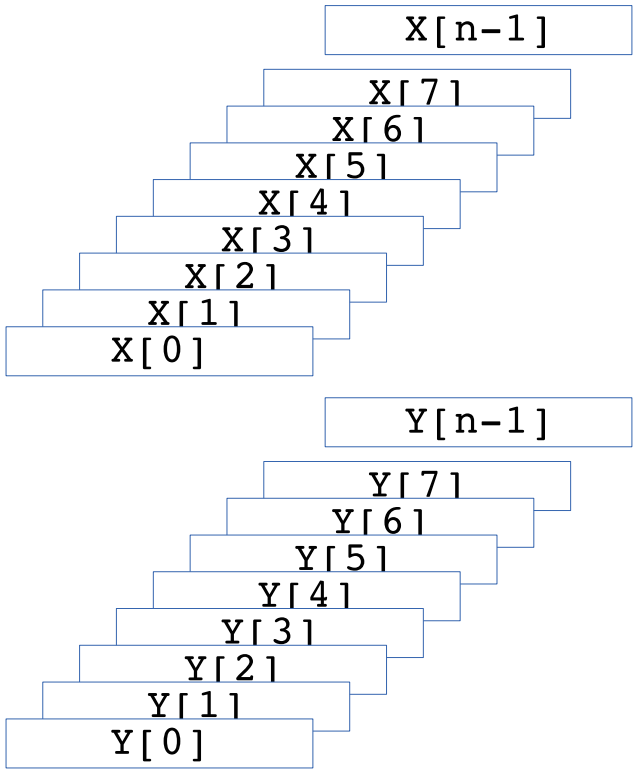
Example 1: SAXPY with SIMD instructions (3/5)

A single instruction allows the four multiplications to be executed: $a \cdot X[i]$.
Another instruction executes the four sums:



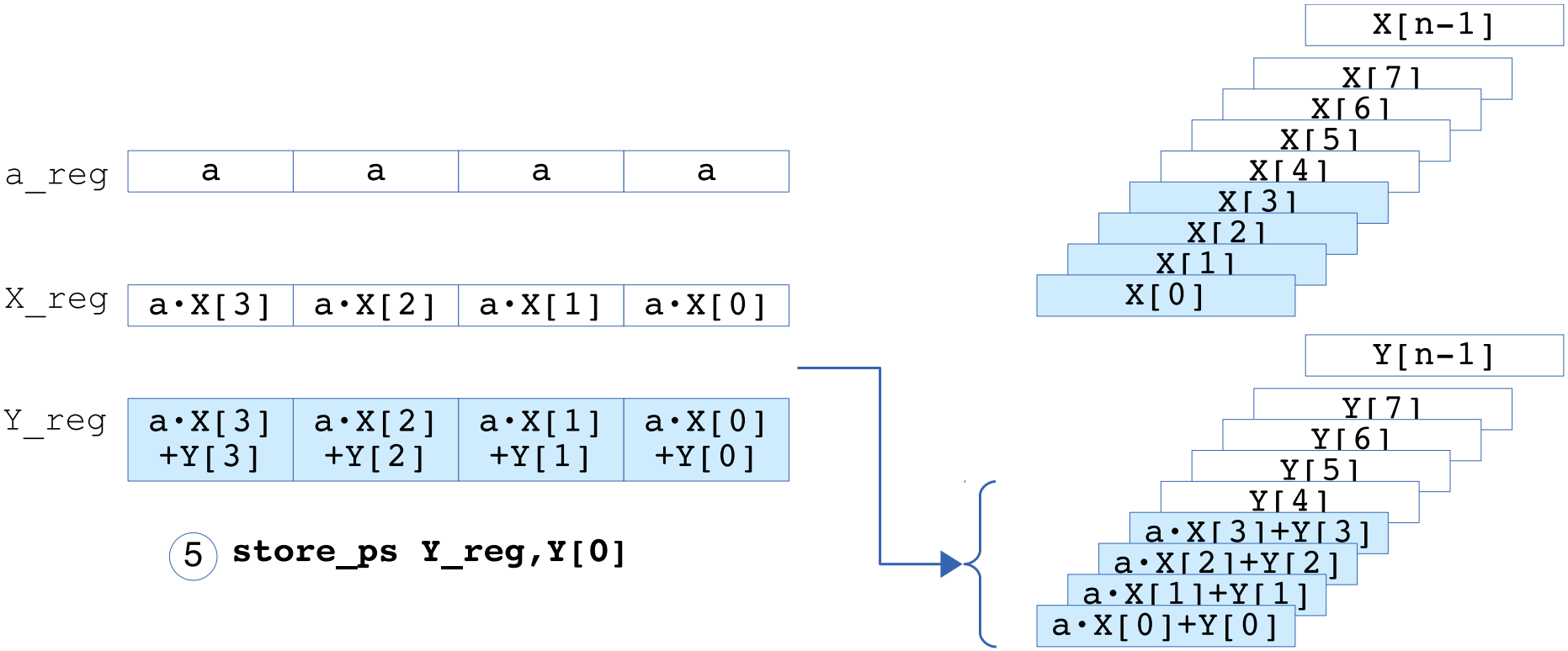
3 **mul_ps X_reg,X_reg,a_reg**

4 **add_ps Y_reg,X_reg,Y_reg**



Example 1: SAXPY with SIMD instructions (4/5)

The fifth instruction updates the first four components $Y[i]$ in memory



Example 1: SAXPY with SIMD instructions (5/5)

A single iteration of the loop with SIMD instructions is equivalent to four iterations with non-vector instructions.

With SIMD instructions

It is necessary to iterate $n/4$ times
Operations: $n/2$ load instructions, $n/4$ multiplication instructions, $n/4$ add instructions, and $n/4$ store instructions

Compiler support

Compilation is based on data types and expressions to generate code.

- If the programmer writes the code in a scalar manner, a classical compiler will not use SIMD instructions

Three ways to insert SIMD instructions into the code:

Automatic vectorization

The programmer does not explicitly describe the parallelism.
An advanced compiler extracts parallelism from scalar source code.

Using SIMD data types

The programmer defines the relevant variables with a specific data type and uses them in common language expressions.

Intrinsic functions

The language has a library of SIMD functions.
The programmer uses them explicitly.

Compilation of example 1 (1/3)

Automatic vectorization

The source code contains no reference to vectorization

```
void SAXPY(int n, float a, float *X, float *Y) {
    int i;
    for(i=0; i<n; i++)
        Y[i] = a*X[i] + Y[i];
}
```

The compiler option -O3

```
gcc -O3 saxpy.c -o saxpy
```

will optimize the code (if possible) by inserting SIMD instructions

Compilation of example 1 (2/3)

Using SIMD data types

The type "`__m128`" allows to specify blocks of 4 elements.

```
void saxpy(int n, float a, float *X, float *Y) {
  __m128 *x_ptr, *y_ptr;
  int i;
  x_ptr = (__m128 *) X;
  y_ptr = (__m128 *) Y;
  for (i=0; i<n/4; i++)
    y_ptr[i] = a*x_ptr[i] + y_ptr[i];
}
```

The compiler infers that the expression "`a*x_ptr[i] + y_ptr[i]`" should be compiled using SIMD instructions.

Compilation of example 1 (3/3)

Intrinsic functions

There are libraries that allow the programmer to insert specific code. The functions "_mm_load_ps", "_mm_add_ps", "_mm_mul_ps" and "_mm_store_ps" translate into appropriate SIMD instructions.

```
void saxpy(int n, float a, float *X, float *Y) {
  __m128 x_vec, y_vec, a_vec;
  int i;
  a_vec = _mm_set1_ps(a);
  for (i=0; i<n; i+=4) {
    x_vec = _mm_load_ps(&X[i]);
    y_vec = _mm_load_ps(&Y[i]);
    x_vec = _mm_mul_ps(a_vec, x_vec);
    y_vec = _mm_add_ps(x_vec, y_vec);
    _mm_store_ps(&Y[i], y_vec);
}
```

Example 2: Vector dot product (1/5)

Conventional code

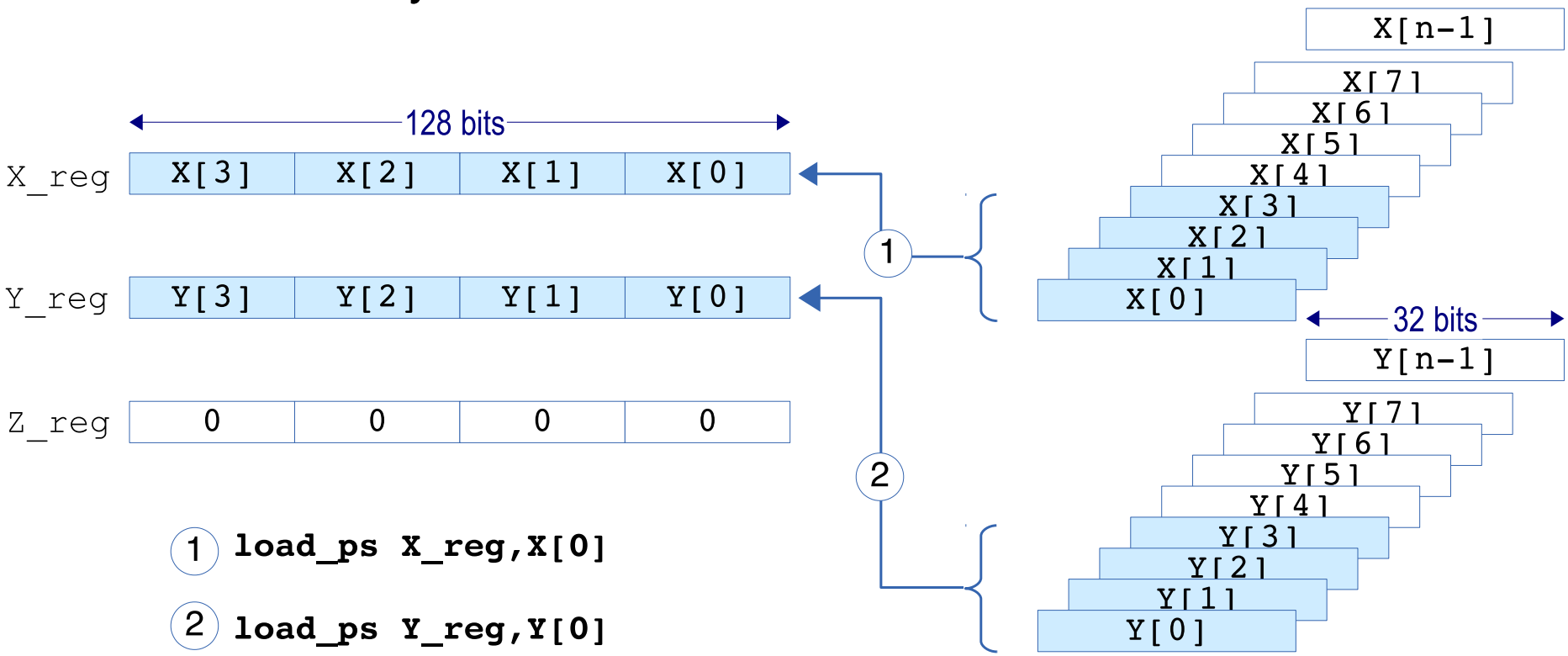
```
float Scalar(int n, float *X, float *Y) {
    int i;
    float prod = 0.0;

    for(i=0; i<n; i++)
        prod += X[i] * Y[i];
    return prod;
}
```

Without SIMD instructions

- *n* iterations,
- *n* multiplications and
- *n* additions.

Example 2: Vector dot product (2/5)

Vectorization: Every iteration will read two blocks...

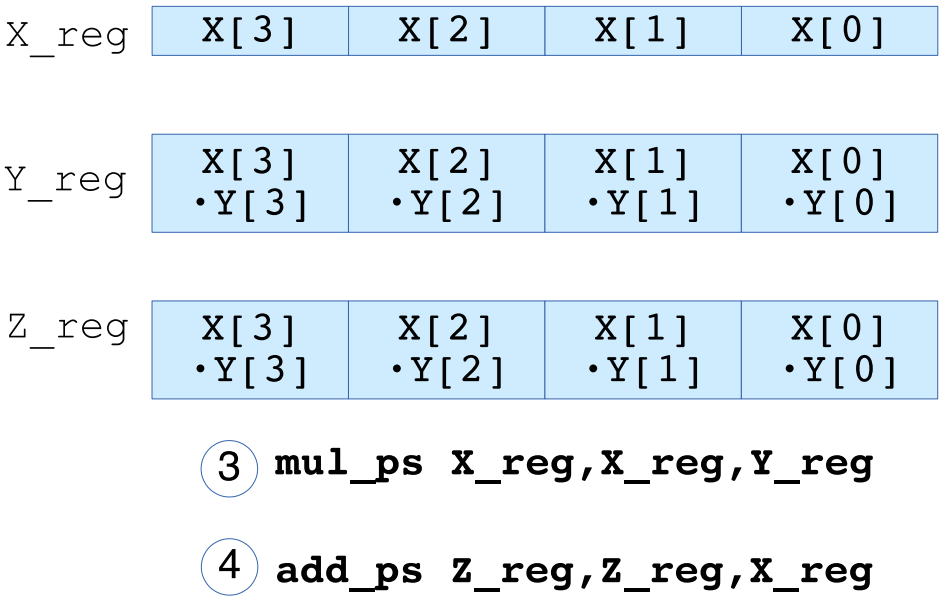


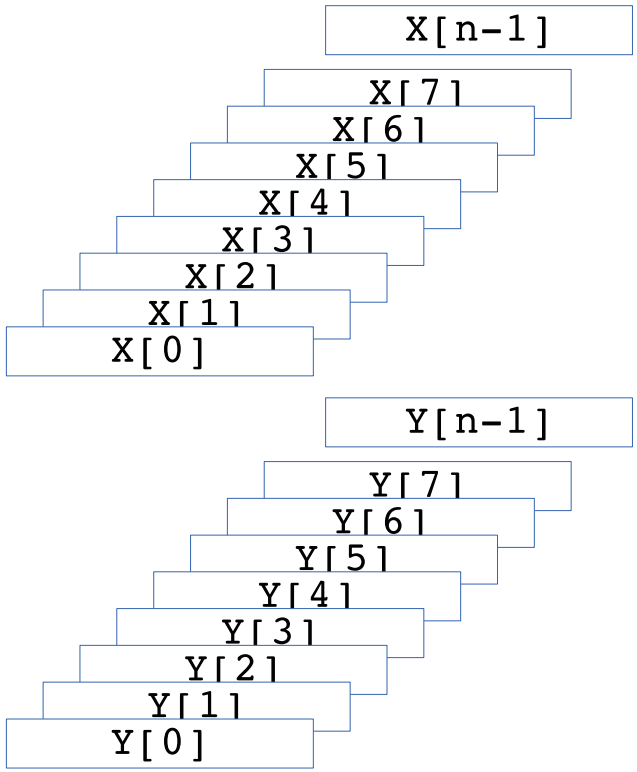
1 **load_ps X_reg,X[0]**

2 **load_ps Y_reg,Y[0]**

Example 2: Vector dot product (3/5)

...and will execute four multiplications and four additions.





Example 2: Vector dot product (4/5)

At the end of the $n/4$ iterations, register Z_reg will hold, for $i = 0 \ldots (\frac{n}{4} - 1)$:
$\sum(X_{4i+3} \cdot Y_{4i+3})$, $\sum(X_{4i+2} \cdot Y_{4i+2})$, $\sum(X_{4i+1} \cdot Y_{4i+1})$ and $\sum(X_{4i} \cdot Y_{4i})$
The dot product is the result of adding the four values.

With SIMD instructions

- $n/4$ iterations,
- $n/4$ multiplications and
- $4 + n/4$ additions.

Example 2: Vector dot product (5/5)

SIMD code

```
float Scalar(int n, float *X, float *Y) {
    float prod = 0.0;
    int i;
    __m128 X_reg, Y_reg, Z_reg;
    Z_reg = _mm_setzero_ps();
    for(i=0; i<n; i+=4) {
        X_reg = _mm_load_ps(&X[i]);
        Y_reg = _mm_load_ps(&Y[i]);
        Y_reg = _mm_mul_ps(X_reg, Y_reg);
        Z_reg = _mm_add_ps(Y_reg, Z_reg);
    }
    for(i=0; i<4; i++)
        prod += Z_reg[i];
    return prod;
}
```