Corso di Architettura dei Sistemi a Microprocessore

Architetture SIMD

Luigi Coppolino

Contact info

Prof. Luigi Coppolino
luigi.coppolino@uniparthenope.it

Università degli Studi di Napoli "Parthenope"
Dipartimento di Ingegneria

Centro Direzionale di Napoli, Isola C4
V Piano lato SUD - Stanza n. 512

Tel: +39-081-5476702
Fax: +39-081-5476777

References

- Textbook: Chapter 11

Chapter Outline

- Multithreading
- Vector (SIMD) processing
- Multimedia Extensions (x86)
- GPU

Hardware Multithreading

- A *process* is a program and its current state (e.g., a Web browser or a word processor)
- Each process has a corresponding *thread*, which is an independent path of execution
- Encompasses state in PC and other registers
- Operating system (OS) enables multitasking by time slicing and context switching of threads
- For efficient handling of multiple threads, processors can use hardware multithreading

Hardware Multithreading

- Hardware includes multiple sets of registers, including multiple program counters
- Each set of registers is for a different thread
- No time is wasted saving/restoring registers
- Context switch simply involves changing a hardware pointer to active set of registers
- *Coarse-grained* multithreading: switch on long-latency events, such as cache misses
- *Fine-grained* or *interleaved*: switch every cycle

Vector (SIMD) Processing

- A *vector* is an array of consecutive elements
- Loops process arrays one element at a time
- Implementing a processor with multiple ALUs enables multiple operations simultaneously
- Requires that programs have *data parallelism*, which means operations are independent
- Use *single-instruction multiple-data (SIMD)* or *vector instructions* and data in *vector registers*

Vectorization

- Vector length *L* for elements per register and number of parallel operations per instruction
- Use this parameter to vectorize loop code
- Involves using vector (SIMD) instructions to reduce the number of instructions executed
- A *vectorizing compiler* can perform analysis to detect data parallelism for transformation
- Each pass of vectorized loop does *L* operations to ideally reduce instruction count by factor *L*

Vector Registers

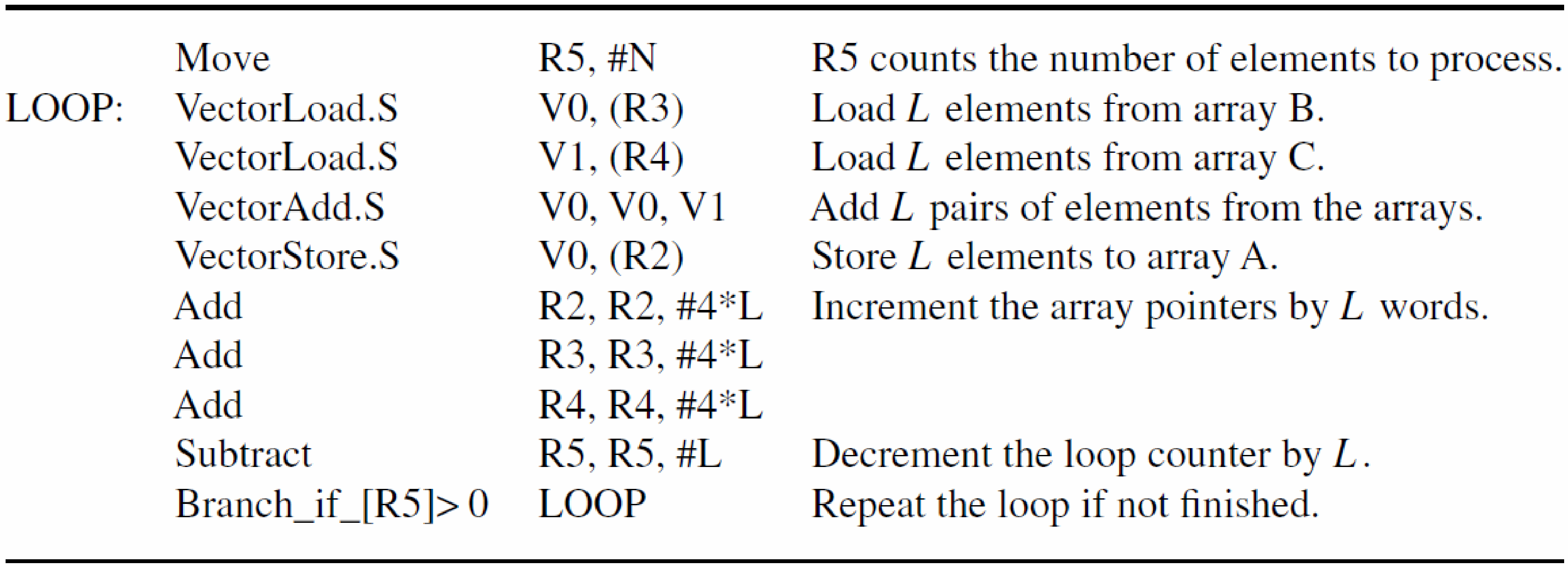
- Vector Instructions data are held in Vector Registers
  - Each Vector Register can store more data elements
  - The number of elements stored in a Vector Register is said Vector Length
- Intel IA-32
  - 128 bit length Vector Regisers
  - L can be from 2 to 16 => data from 64 to 8 bits

```
for (i = 0; i <  N; i++)
    A[i] = B[i] + C[i];
```

(a) A C-language loop to add vector elements

```
         Move               R5, #N        R5 is the loop counter.
LOOP:    Load               R6, (R3)      R3 points to an element in array B.
         Load               R7, (R4)      R4 points to an element in array C.
         Add                R6, R6, R7    Add a pair of elements from the arrays.
         Store              R6, (R2)      R2 points to an element in array A.
         Add                R2, R2, #4    Increment the three array pointers.
         Add                R3, R3, #4
         Add                R4, R4, #4
         Subtract           R5, R5, #1    Decrement the loop counter.
         Branch_if_[R5]> 0  LOOP          Repeat the loop if not finished.
```

(b) Assembly-language instructions for the loop



```
       Move               R5, #N        R5 counts the number of elements to process.
LOOP:  VectorLoad.S       V0, (R3)      Load L elements from array B.
       VectorLoad.S       V1, (R4)      Load L elements from array C.
       VectorAdd.S        V0, V0, V1    Add L pairs of elements from the arrays.
       VectorStore.S      V0, (R2)      Store L elements to array A.
       Add                R2, R2, #4*L  Increment the array pointers by L words.
       Add                R3, R3, #4*L
       Add                R4, R4, #4*L
       Subtract           R5, R5, #L    Decrement the loop counter by L.
       Branch_if_[R5]> 0  LOOP          Repeat the loop if not finished.
```

(c) Vectorized form of the loop

Streaming SIMD Extensions (SSE)

- Introduced by Intel with Pentium III
  - It is an evolution of the MMX technology previously implemented by Intel
  - SSE was introduced as a reply to the AMD 3DNow! Implementation of the MMX technology
    - Later AMD implemented the SSE instruction set
- Is a SIMD extention of the Pentium ISA
- SSE makes use of 8 vectored registers (128 bits) for floating point operations
  - SSE2 extends the usage of vectored registers to integer operations
- With SSE2 it was possible use the vector registers to store:
  - two 64-bit double-precision floating point numbers or
  - two 64-bit integers or
  - four 32-bit integers or
  - eight 16-bit short integers or
  - sixteen 8-bit bytes or characters.
- Pentium IV introduced SSE3 and in its last implementation SSE4

Multimedia Extensions Vs Vector Processing

- SSE vector registers are typically much shorter than vector processors registers
- The size of the operands is encodend in the opcode for SSE while it is in a separate register for VPs
- A vector is loaded with consecutive memory locations in SSE while this is not a constraint for VPs
    - As an example it can be loaded with elements at a fixed distance

Graphics Processing Units (GPUs)

- Demand for high-resolution, 3-D graphics has driven development of powerful GPU chips
- Graphics applications have data parallelism in floating-point calculations and other tasks
- Large GPUs have hundreds of simple cores
- General vector processing possible with cores executing the same program on different data
- Programming support: CUDA C and OpenCL