An Introduction to Parallelism in i860™ Microprocessor

Yingchun Wu
ahwu@warp.hut.fi
Helsinki University of Technology
Institute of Photogrammetry and Remote Sensing
02150 Espoo, Finland

Abstract

The i860 is a single chip 64-bit RISC microprocessor. It is being used in a range of applications such as engineering workstations, scientific computing, 3D graphics workstations, numeric accelerators, and multi-user and multiprocessor system. The RISC architecture and intrinsic parallelism make i860 suitable for massive data processing, especially for image processing. It is necessary and important to master the i860 processor's capabilities and parallelism for setting up an i860-based system applied for massive data processing.

1. Introduction

The i860 is a 64-bit, RISC(Reduced Instruction Set Computer), Intel microprocessor that can perform up to 80 MFlops(Millions of Floating-point Operations Per Second) and 40 MIPS(Millions of Instructions Per Second) at 40 Mhz [4]. Its advanced architecture concepts and parallel techniques make i860 acting an important role in multiprocessor system for parallel data processing.

There were some multiprocessor systems based on i860 for parallel data processing. For instance, MFDSM (MultiFunction Distributed Shared Memory Architecture) used i860 microprocessor to develop a multiprocessor system for image processing in photogrammetric area[2].

Some tests were taken for the performance of iPSC/860 (Intel i860-based hypecube). T.H. Dunigan compared the iPSC/860 and Ncube 6400 (New Ncube hypecube) with their earlier version iPSC/1 (Intel 80286/287-based), iPSC/2 (Intel 80386/387 based) and Ncube 3200 [1]. The Ncube 6400 and iPSC/860 showed marked performance improvements over the earlier hypecubes, providing an increase in both communication and computation speeds and offering increased memory capacity and high-speed routing. The iPSC/860 indicates more effective performances than Ncube 6400. For example an aggregate implementation for a 1000*1000 matrix using 16 nodes was 45.8 MFlops for iPSC/860 compared with 7.4 MFlops for Ncube 6400. As another test of a parallel, double precision, FORTRAN implementation of SLALOM (a program to solve a

radiosity problem that includes file I/O) ran at 15.6 MFlops on a 64-node Ncube 6400 and at 134.8 MFlops on a 64-node iPSC/860. I.Gutheil performed parallel matrix-matrix-multiplication routine DGEMMP on 32-node iPSC/860, which achieved to maximum of 724 MFlops[3].

The high speed of processing in i860-based systems is mainly contributed by the intrinsic parallelism of i860. The image processing tasks are computationally intensive and inherently parallel. Parallel computing is essential to solve such a problem. So it is very vital to have a good grasp of the parallelism if you want to set up an efficient system based on the i860 microprocessor.

The organisation of this paper is as follows. In chapter 2, overview of the i860's architecture is given. The parallelism of i860 -- pipe-lined, dual-operation and dual-instruction mode -- and some examples are explained in chapter 3. Finally the conclusion is drawn in Chapter 4.

2. Architecture of i860 microprocessor

The i860 processor is a single chip, 64-bit microprocessor that balances integer, floating-point, and graphics performance. It is being used in a variety of configurations such as engineering workstations, scientific computing, 3D graphics workstations, numeric accelerators, and multi-user and multiprocessor system.

The major functional unit of the i860 processor and the paths between them are shown in Figure 1 [4]. The processor consists of instruction cache, memory management unit, data cache, bus control unit, RISC core, floating-point control unit, 3D graphics unit, pipelined floating-point adder, and pipelined floating-point multiplier. The wide information paths include 64-bit external data bus, 128-bit on-chip data bus, and 64-bit on-chip instruction bus.

The on-chip 4K instruction cache and 8K data cache provide fast access to the most recently referenced memory locations. The on-chip caches and very wide buses balance the data and instruction bandwidth to the processor speed of multiple execution units. Data and instructions are fetched simultaneously by separate instruction and data paths. The data cache connects to the floating-point unit via a 128-bit bus and to the RISC core by a 32-bit bus. Externally, the processor has 32-bit address and a 64-bit data bus, which is used to access external memory.

The i860 processor supports several forms of parallel execution. First, the RISC core, floating-point unit, and graphics unit are pipelined. In pipeline mode, the i860 processor overlaps execution of sequential instructions. Secondly, The instruction cache can fetch two 32-bit instructions simultaneously -- one for execution by the RISC core and one for execution by the floating-point unit. Both the RISC core and the floating-point unit can execute and produce results concurrently. That is dual-instruction mode. Also, within the floating-point unit , the adder and the multiplier unit can be programmed to work together to perform common computations, which is called as dual-operation mode. By combining the floating-point architecture with the graphics unit, it is possible to support interactive visualisation and display pictures and data in the form of three-dimension. The on-chip MMU (Memory Management Unit) provides operating system support for multi-user and multitasking operating systems.

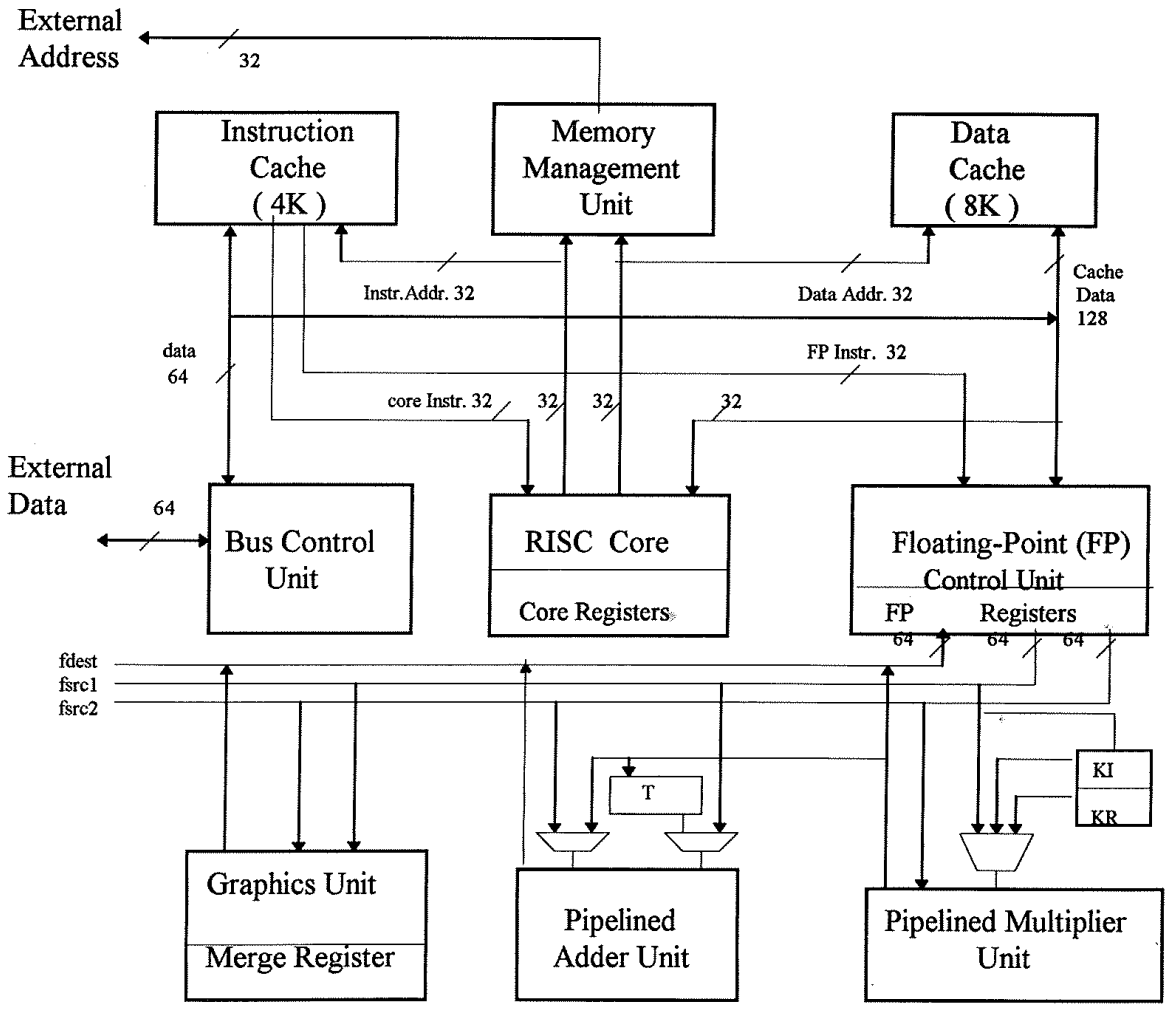


Figure 1 Functional unit and data paths of the i860 microprocessor

3. Parallelism in i860 processor

There are three techniques that can exploit the parallelism available within the i860 microprocessor. The most basic form of parallelism on the i860 microprocessor is the pipelined execution with the goal of reducing the average number of clock per instruction. The second type of parallelism is the dual-operation mode that performs the adder and multiplier in parallel. The final type of parallelism is a dual-instruction mode where both the RISC core and the floating-point unit execute simultaneously.

3.1 Pipelined Mode

The concept of pipeline processing in a computer is similar to assembly lines in an industrial plant in order to increase productivity. Their original form is a flow line (pipeline) of assembly stations where items are assembled continuously from separate parts along the moving conveyor belt. In the i860, pipelining is designed for reducing the average number of clocks per instruction through overlapping the executions of multiple instructions.

Modern RISC processors break down instruction execution steps into simple stages. RISC instruction sets move efficiently through the processor's pipeline by taking only one clock cycle at each stage. For example, pipelined FP(Floating-Point) add instruction is divided into 3 stages that takes one clock cycle at each stage. RISC processors not only define their instructions to be easily broken up into simple execution stages, but also allow instruction to start with each clock cycle by overlapping the instruction. With pipelined instructions, the adder can produce a result every clock cycle, and the multiplier can produce a result every clock for single-precision and every other clock cycle for double-precision. On the other hand, the normal scalar instructions take three clock cycle to complete a FP add or subtract operation, and three clock cycles for single-precision and four clock cycles for double-precision in a multiply operation.

Figure 2 illustrates an example to compare the scalar instructions and the pipelined instructions summing two vectors with 7 elements each. Vector A is {2.0,3.0,4.0,5.0,6.0,7.0} which are stored in FP registers from **f2** to **f7**. Vector B is {8.0,9.0,10.0,11.0,12.0,13.0} which are stored in FP registers from **f8** to **f13**. First we use scalar instruction to add elements of two vectors. The sequence of instructions is shown in Figure 2a. The whole process takes 18 clock cycles, because it takes 3 clock cycles for each scalar instruction execution.

The pipelined instruction and adder pipeline stages are demonstrated in Figure 2b. There are 3 stages in adder pipeline as well as in multiplier pipeline. Pipelined instructions advance one stage of pipeline every clock cycle. The instruction starts as soon as the previous instruction enters the next stage and at the same time the result of the previous last stage is stored to the destination specified by the instruction which just entered the first stage.

Instruction Sequence :

```
fadd.ss     f2, f8 , f14        ; f14 ← f2 +f8
fadd.ss     f3, f9 , f15        ; f15 ← f3 +f9
fadd.ss     f4, f10, f16        ; f16 ← f4 +f10
fadd.ss     f5, f11, f17        ; f17 ← f5 +f11
fadd.ss     f6, f12, f18        ; f18 ← f6 +f12
fadd.ss     f7, f13, f19        ; f19 ← f7 +f13
```

a. Scalar floating-point addition

Instruction Sequence	Adder Pipeline stages			Destination
// initial phase				
pfadd.ss f2 , f8 , f0	2+8	?	?	none
pfadd.ss f3 , f9 , f0	3+9	2+8	?	none
pfadd.ss f4 , f10, f0	4+10	3+9	2+8	none
// continuous phase				
pfadd.ss f5 , f11, f14	5+11	4+10	3+9	f14 ← 10
pfadd.ss f6 , f12, f15	6+12	5+11	4+10	f15 ← 12
pfadd.ss f7 , f13, f16	7+13	6+12	5+11	f16 ← 14
// draining phase				
pfadd.ss f0 , f0 , f17	0	7+13	6+12	f17 ← 16
pfadd.ss f0 , f0 , f18	0	0	7+13	f18 ← 18
pfadd.ss f0 , f0 , f19	0	0	0	f19 ← 20

b. Pipeline floating-point addition

Figure 2 Adder scalar and pipeline instruction example

To execute pipelined instructions, we have to initialise the adder pipeline stage by first three instructions. Therefore, the result is sent to **f0**, which is always zero and is used as a null destination. On each successive clock cycle, a **pfadd** instruction (see Appendix A on i860 instructions for details) advances the pipeline. When the result of the first **pfadd** becomes available at the last stage, it is stored to the destination specified by the forth instruction. From the forth instruction, there is one result produced every clock cycle. And of course it needs 3 pipelined instructions to drain the adder pipeline stages. So it takes 9, i.e. 3+6-3+3 clock cycles, to sum those two vectors by pipelined instructions.

Vector processing problems are solved most effectively by pipelined instruction. Suppose to calculate the summation of two n-element vectors. 3*n clock cycles are taken in scalar mode, while 3+n clock cycles are cost in pipelined mode. If n is large enough, the pipelined mode uses one-third time of scalar mode's. The more the repetitive operations that do not depend on the result of the previous operation are, the less time the pipeline mode takes than scalar mode does.

3.2 Dual-operation Mode

Dual-operation mode is another distinguishing feature in i860 parallelism. The adder unit and multiplier unit are both issued in parallel with dual-operation instruction.

Figure 3 shows the adder unit, multiplier unit, special registers, and the data paths in dual-operation mode. A dual-operation instruction requires for six operands, two inputs and one output for both the adder and multiplier.

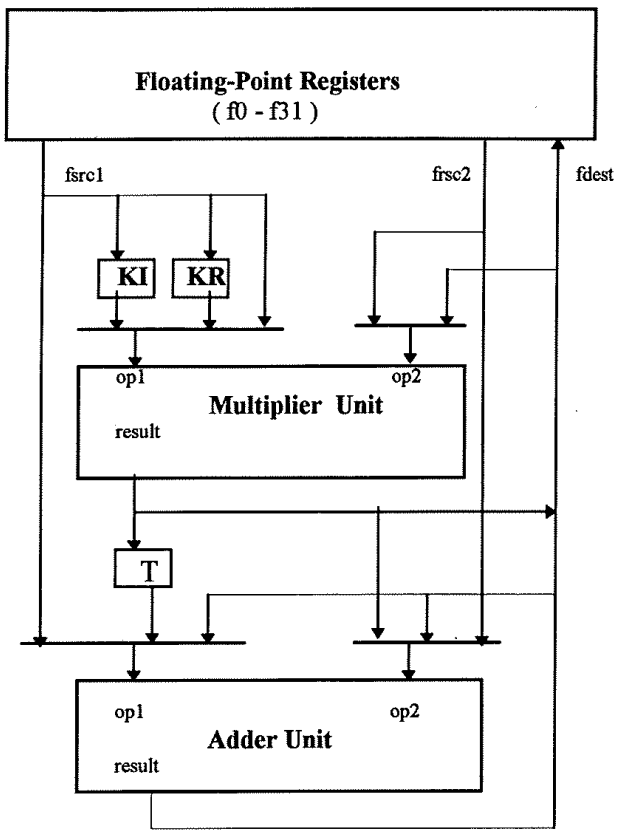


Figure 3 Floating-point unit and dual-operation data paths

Register KI, KR, T can be used to store and subsequently supply data for dual-operation instructions. The source and destination paths for the units are specified by the instruction encoding. There are 31 different instructions in dual-operation mode.

For single-precision, one result per clock cycle can be produced by both the adder and multiplier units, reaching a peak rate of 80 MFlops at 40 MHz. For double-precision, the multiplier can produce a result every other clock cycle, and the adder produces a result every clock cycle, for a peak rate of 60 MFlops.

The following example explains how the dot product of two eight-element vector A and B is produced with pipelined dual-operation instructions.

Assume that the actual matrices to be multiplied have the following values:

A = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0] ;

B = [8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0]^{T};

Suppose that the vector A and B are already loaded into the registers from **f4** to **f11** and from **f12** to **f19** respectively. The operation to calculate is 1.0*8.0+2.0*7.0+...+8.0*1.0, a series of multiplication followed by additions. At the heart of the code that is arranged as Figure 5 is a dual-operation instruction **m12apm.** Before going further, Figure 4 plots the structure of adder and multiplier units for the instruction **m12apm**. The first and second operand of the instruction multiply in multiplier pipeline. At the same time the result of multiplier pipeline's last stage becomes a first operand of the adder and the result of adder pipeline's last stage enters the adder as a second operand . The result of the last adder pipeline stage is stored in the destination.

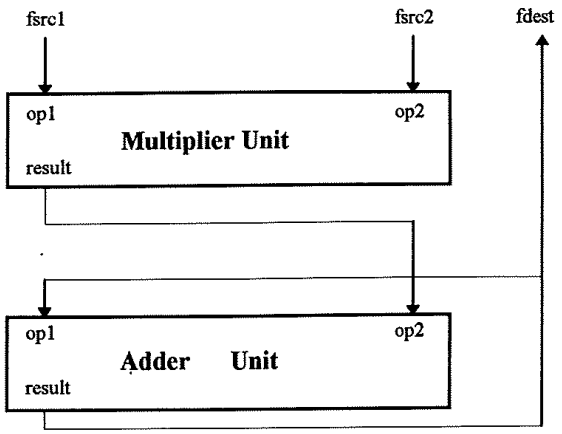
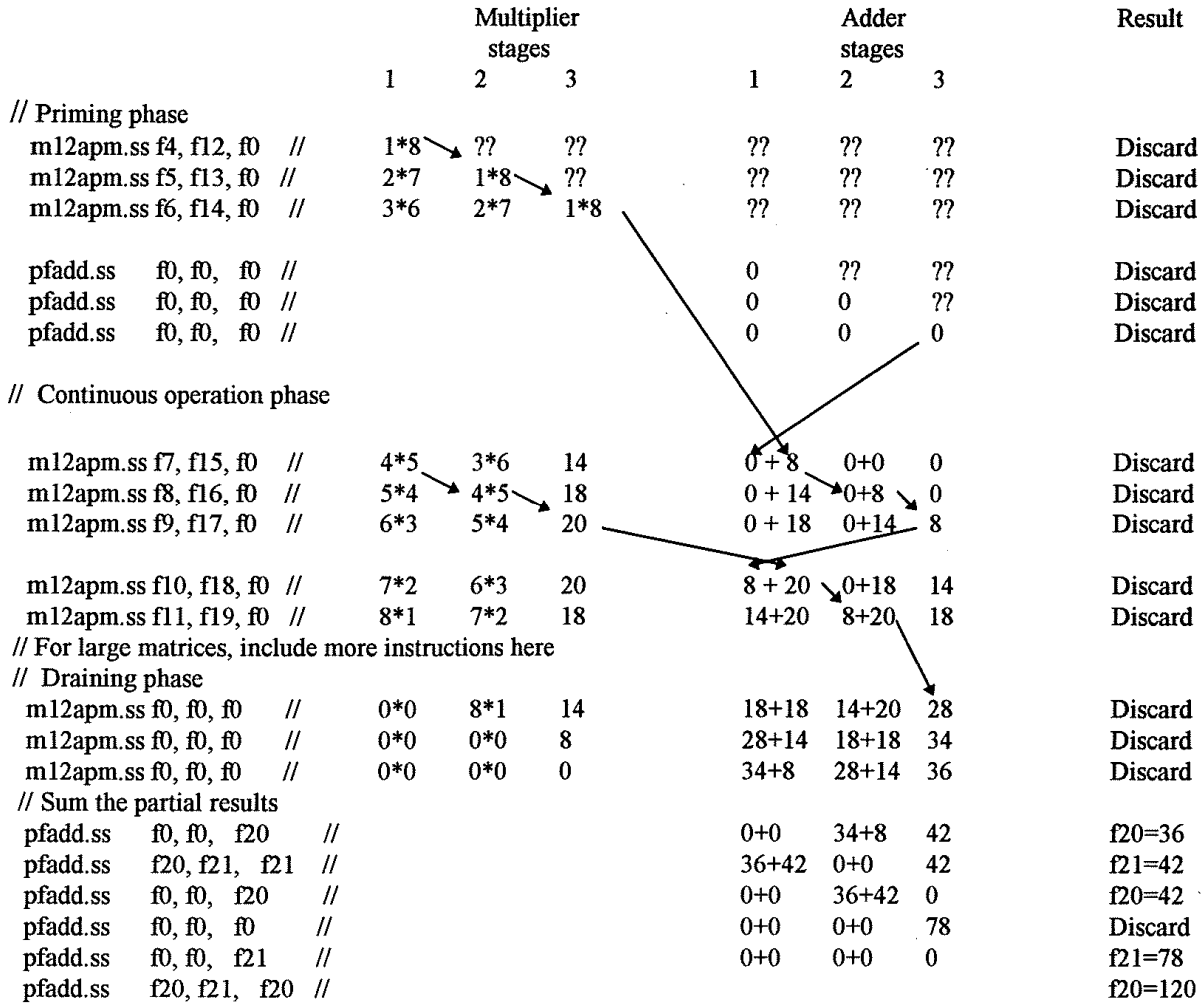


Figure 4 Data paths of instruction m12apm

The code list is divided into three phases: priming, continuous operation, and draining phase. The priming phase consists of three **m12apm** instructions to load the multiplier stages and discards the results and three **pfadd** instructions to clear the adder pipeline. In continuous phase, **m12apm** instructions perform the multiplication, feed the intermediate results to the adder, and execute the add operation to accumulate the products. The draining phase finally retrieves the intermediate results and produces the final dot-product result.



	Multiplier stages			Adder stages			Result
	1	2	3	1	2	3	
// Priming phase							
m12apm.ss f4, f12, f0 //	1*8	??	??	??	??	??	Discard
m12apm.ss f5, f13, f0 //	2*7	1*8	??	??	??	??	Discard
m12apm.ss f6, f14, f0 //	3*6	2*7	1*8	??	??	??	Discard
pfadd.ss f0, f0, f0 //				0	??	??	Discard
pfadd.ss f0, f0, f0 //				0	0	??	Discard
pfadd.ss f0, f0, f0 //				0	0	0	Discard
// Continuous operation phase							
m12apm.ss f7, f15, f0 //	4*5	3*6	14	0 + 8	0+0	0	Discard
m12apm.ss f8, f16, f0 //	5*4	4*5	18	0 + 14	0+8	0	Discard
m12apm.ss f9, f17, f0 //	6*3	5*4	20	0 + 18	0+14	8	Discard
m12apm.ss f10, f18, f0 //	7*2	6*3	20	8 + 20	0+18	14	Discard
m12apm.ss f11, f19, f0 //	8*1	7*2	18	14+20	8+20	18	Discard
// For large matrices, include more instructions here							
// Draining phase							
m12apm.ss f0, f0, f0 //	0*0	8*1	14	18+18	14+20	28	Discard
m12apm.ss f0, f0, f0 //	0*0	0*0	8	28+14	18+18	34	Discard
m12apm.ss f0, f0, f0 //	0*0	0*0	0	34+8	28+14	36	Discard
// Sum the partial results							
pfadd.ss f0, f0, f20 //				0+0	34+8	42	f20=36
pfadd.ss f20, f21, f21 //				36+42	0+0	42	f21=42
pfadd.ss f0, f0, f20 //				0+0	36+42	0	f20=42
pfadd.ss f0, f0, f0 //				0+0	0+0	78	Discard
pfadd.ss f0, f0, f21 //				0+0	0+0	0	f21=78
pfadd.ss f20, f21, f20 //							f20=120

Figure 5 Dot-product with pipelined dual-operation instruction (single-precision)

3.3 Dual-instruction Mode

To further achieve peak performance for the inner loops of common routines, the i860 processor supports DIM (Dual-Instruction Mode). In Dual-instruction mode, RISC core and floating-point operation perform loads and stores to the floating-point registers simultaneously. The RISC core fetches two 32-bit instructions per clock using the 64-bit wide instruction cache. Of the instructions fetched, one is executed by RISC core and one by floating-point unit. The RISC core keeps the data flowing to the floating-point processing units by fetching and storing information to the floating-point registers, while the floating-point execution units are operating on the data already in the registers. In dual-instruction mode, each clock cycle of execution consists of instruction pairs of a floating-point and an integer instruction. The floating-point instruction may be one of the dual-operation instructions, for a total of three operations per clock cycle.

Figure 6 illustrates the sequence for entering and exiting dual-instruction mode. When the i860 is executing in single-instruction mode and encounters a floating-point instruction with the d-bit set,

such as instruction **d.pfadd.ss** with the first letter "d", one more 32-bit instruction is executed before dual-instruction mode begins. To continue performing in dual-mode, each floating-point instruction must have the d-bit set. If a floating instruction is encountered with d-bit clear in dual-mode, then one more pair of instructions is executed before changing to single-mode.

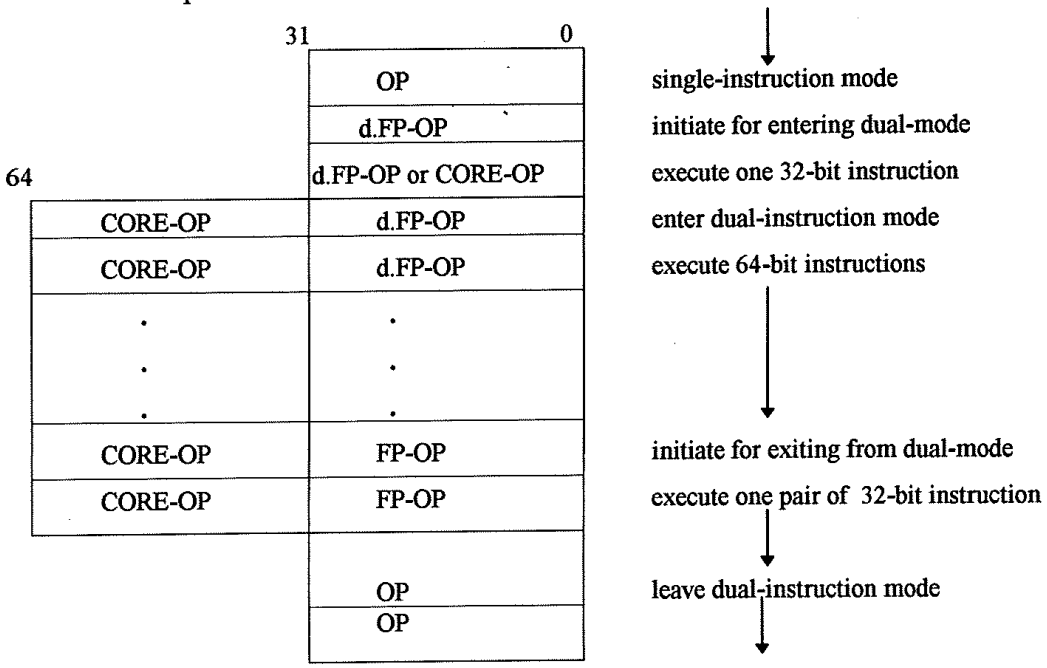


Figure 6 Dual-instruction mode transition

Vector calculations involve repeating the same operation on a series of data values. Although there are no specific vector instructions on the i860 processor, the parallelism of the architecture can support traditional vector processing.

Consider a simple problem to add the single-precision elements of an arbitrarily long vector. The procedure uses the basic pipelined floating-point add instruction **pfadd.ss** in dual-instruction mode to overlap loading, decision making, and branching shown in Figure 7 [5].

The program starts by loading the first two vector elements for the first pass of loop 1, initialising the counter variable. Clear the first stage of adder pipeline and decrease the size of vector before entering dual-instruction mode. While entering dual-instruction mode, clear the rest stage of the adder pipeline, initialise Loop-Condition-Code, and load the third and forth vector elements for the first pass of loop 2. The summation loop is actually implemented as two loops marked with the starting labels **L1** and **L2**. The loops are executed entirely in dual-instruction mode.

The first loop, **L1**, adds **f20** and **f21** to the running totals in **f30** and **f31,** and loads new elements to the **f20** and **f21** registers with a single **fld.d** instruction. If there are additional elements, the program jumps to the second loop, **L2.** Loop 2 adds **f22** and **f23** to running totals in **f30** and **f31**, and loads new elements into the **f22** and **f23** registers. Execution continues to alternate between loop 1 and loop 2 until there are no more elements to be loaded. Note that the instruction pair following the **bla** instruction is executed regardless of the branch decision.

When the loop count is reached, execution falls through the loops continuing to sum the elements already loaded into registers and loads the final elements. When the label **S** is reached, execution is returned to single-instruction mode and, if the number of elements is odd, one more load and

addition are performed. The code denoted by the label **DONE** performs the summation total into **f16**.

```
// input  : r16  -- vector address,  r17 -- vector size (must be > 5)
// output : f16  -- sum of vector elements
        fld.d       r0(r16),        f20     // load first two elements
        mov         -2,             r21     //loop decrement for bla
// initiate entry into dual-instruction mode
        d.pfadd.ss  f0,     f0,     f0      // clear adder pipeline (stage 1)
        adds        -6,     r17,    r17     //decrement size by 6
// enter into dual-instruction mode
        d.pfadd.ss  f0,     f0,     f0      // clear adder pipeline  (stage 2)
        bla         r21,    r17,    L1      // initialise LCC
        d.pfadd.ss  f0,     f0,     f0      // clear adder pipeline (stage 3)
        fld.d       8(r16)++,       f22     //loading 3rd and 4th elements
L1::    d.fadd.ss   f20,    f30,    f30     // add f20 to pipeline
        bla         r21,    r17,    L2      // If more , go to L2 after
        d.pfadd.ss  f21,    f31,    f31     // adding f21 to pipeline and
        fld.d       8(r16)++,       f20     // loading next f20:f21
// If we reach this point, at least one element remains to be loaded.
// r17 is either -4 or -3.
// f20,f21,f22, and f23 still contain vector elements.
// add f20 and f22 to pipeline, too.
        d.pfadd.ss  f20,    f30,    f30
        br          S                       // exit loop after adding
        d.pfadd.ss  f21,    f31,    f31     // f21 to pipeline
        nop
L2::    d.pfadd.ss  f22,    f30,    f30     //add f22 to pipeline
        bla         r21,    r17,    L1      // if more, go to L1 after
        d.pfadd.ss  f23,    f31,    f31     //adding f23 to pipeline and
        fld.d       8(r16)++,       f22     //loading next f22:f23
// If we reach this point, at least one element remains to be loaded.
// r17 is either -4 or -3.
// f20,f21,f22, and f23 still contain vector elements.
// add f20 and f22 to pipeline, too
        d.pfadd.ss  f20,    f30,    f30
        nop
        d.pfadd.ss  f21,    f31,    f31
        nop
S::     //initiate exit from dual-mode
        pfadd.ss    f22,    f30,    f30     // still in dual-mode
        mov         -4,             r21
        pfadd.ss    f23,    f31,    f31     //last dual-mode pair
        bte         r21,    r17,    DONE    // if there is one more element,
        fld.l       8(r16)++,       f20     // load it and
        pfadd.ss    f20,    f30,    f30     //add it to pipeline
// intermediate results are sitting in the adder pipeline,
// let A1:A2:A3 represent the current pipeline contents.
DONE:: pfadd.ss     f0,     f0,     f30     // 0        : A1    : A2    f30=A3
       pfadd.ss     f30,    f31,    f31     // A2+A3    : 0     :A1     f31=A2
       pfadd.ss     f0,     f0,     f30     // 0        : A2+A3: 0      f30=A1
       pfadd.ss     f0,     f0,     f0      // 0        : 0     :A2+A3
       pfadd.ss     f0,     f0,     f31     // 0        : 0     :0      f31=A2+A3
       fadd.ss      f30,    f31,    f16     // f16=A1+A2+A3
```

Figure 7 Vector sum in dual-instruction mode

3. Conclusion

The i860 microprocessor brings new levels of performance and capability to the microprocessor world. The RISC architecture and several parallelism inside the processor give the processor a high speed in executing, flexibility in programming, and wide field in applying.

Many algorithms such as matrices operations, Fast Fourier Transform (FFT) [4], 3D graphics transform [4] can be performed in very fast way in i860 processor with its parallelism. For instances, the butterfly algorithm which is the inner function of FFT runs at 13.3 MFlops in scalar way, 17.4 MFlops in pipelined mode, 23.5 MFlops in dual-instruction mode, 26.7 MFlops in dual-instruction and dual-operation mode, finally reaches 57.1 MFlops in optimal pipelined dual-operation and dual-instruction mode[4]. The evolution of butterfly implementation produces a speed-up of over four times from simplest scalar code to a top-performance parallel version.

Pipeline mode, dual-operation mode, and dual-instruction mode are the characteristics in i860 microprocessor design. Dual-instruction mode allows a memory fetch or store simultaneously with the multiply and add. The floating-point adder and multiplier can produce one sum and one product per clock cycle currently in dual-operation mode. Pipelined instruction issues one result every clock cycle after 3 initialising instructions for each pipeline. The features of i860 microprocessor can be exploited in programming an effective implementation. The advantages in RISC architecture and parallelism bring i860 microprocessor a promising future in multiprocessor system in many application fields.

4. Acknowledgement

The author would like to acknowledge Professor Henrik Haggrén and Anita Laiho for their strongly supports to the research work. The author would like to thank her colleague Min Gong for his many helpful discussions and valuable advice.

References

1. T.H. Dunigan, Performance of the Intel iPSC/860 and Ncube 6400 hypecubes, Parallel computing ,vol. 17 (1991), no.10&11, 1285-1302.
2. M. Gong, Attempt to solve memory access conflict problem in multiprocessor environment MultiFunction Distributed Shared Memory Architecture, SPIE 1991, Boston, USA
3. I. Gutheil, W. Krotz-Vogel, Performance of a parallel matrix multiplication routine on Intel iPSC/860, Parallel computing, vol.20(1994), no.7, 953-974.
4. N. Margulis, i860™ Microprocessor architecture, 1990, McGraw-Hill, Inc.
5. i860™ Microprocessor family programmer's reference manual, 1991, Intel Corporation.

Appendix A i860 instructions that are mentioned in the paper.

adds isrc1, iscr2, idest .....add signed
idest ⟵ isrc1+isrc2
OF ⟵ (bit 31 carry <> bit 30 carry)
CC set , if isrc2 + isrc1 < 0 (signed) , cleared otherwise

bla isrc1ni,isrc2, sbroff .....branch on LCC and add
LCC-temp clear if isrc2 + isrc1ni < 0 (signed), LCC-temp set otherwise
isrc2 ⟵ isrc1ni + src2
execute one more sequential instruction
if LCC then LCC ⟵ LCC-temp, continue execution at brx(sbroff)
else LCC ⟵ LCC-temp
fi

```
br      lbroff ..... branch direct unconditionally
        Execute one more sequential instruction
        continue execution at brx(lbroff)

bte     isrc1s,  isrc2,   sbroff  .....branch if equal
        if       isrc1s  =  isrc2
        then   continue execution at brx( sbroff )
        fi

fadd.p  fsrc1,fsrc2, fdest .....floating-point add
        fdest   <----   fsrc1+fsrc2

fld.y   isrc1(src2), fdest .....(normal)
fld.y   isrc1(isrc2)++, fdest .....(autoincrement)
        fdest   mem.y(src1+isrc2)
        if      autoincrement
        then    isrc2   <----   isrc1 + isrc2
        fi

mov     const32,idest .....constant-to-register move
        if      0x8000 < const32 < = 0xFFFF8000
        then    add  l%const32 , r0,idest
        else    orh h%const32, r0, idest ,
                or  l%const32, idest,dest
        fi

m12apm.p fsrc1, fsrc2, fdest  ..... pipelined floating-point multiply with add
        fdest   last stage of multiplier result
        M-op1   <----   fsrc1
        M-op2   <----   fsrc2
        A-op1   <----   A-result
        A-op2   <----   M-result
        advance A and M pipeline one stage
        A pipeline first stage   <----   A-op1 +A-op2
        M pipeline first stage   <----   M-op1 * M-op2

nop     .....core-unit no operation

or      isrc1, isrc2, idest .....logical OR
        idest   <----   isrc1 OR isrc2
        CC set if  result is zero , cleared otherwise

orh     #const, isrc2, idest ..... logical OR high
        idest   <----   (#const shifted left 16 bits) OR isrc2
        CC set if  result is zero. cleared otherwise

pfadd.p fsrc1, fsrc2, fdest ..... pipelined floating-point add
        fdest   <----   last stage adder result
        Advanced A pipeline one stage
        A pipeline first stage   <----   fsrc1 +fsrc2
```

XVIII ISPRS-Congress Vienna, 9 -19 July 1996

Spatial Information from Images

For its XVIII quadrennial Congress, the International Society for Photogrammetry and Remote Sensing (ISPRS) announces a

Call for Papers

If you are engaged in the fields of

Photogrammetry
Remote Sensing
Geo-Information Systems
or any other related area

and if you feel your recent work needs presenting to an international audience, then we invite you to prepare a paper for this major ISPRS Congress in Vienna, Austria. For your orientation, the following list shows a rough overview of the topics covered by the conference's ***80 technical sessions***, ***32 interactive sessions*** and a number of ***inter-disciplinary sessions***:

- Sensors, platforms and imagery
- Photoscanners and quality analysis
- Photogrammetric systems and advances in automation
- Digital image workstations
- Sensor and image orientation
- Integrating GPS into photogrammetry
- Matching and 3-D object restitution
- Scene analysis and machine vision
- Digital elevation models and their applications
- Archaeological, architectural, medical, and other close-range photogrammetry
- Mapping and planning technologies
- Theory, systems and applications of GIS
- 3-D databases and information systems
- Problems of data fusion
- Remote sensing techniques and applications
- Landuse and disaster assessment
- Environmental and global monitoring
- Educational and training matters
- International cooperation, consulting and technology transfer
- Inter-disciplinary topics to AARS, CIPA, IUSM, and OEEPE

All you need to do is write an ***extended abstract*** of 750 words minimum to 1500 words maximum and submit it ***before 16 October 1995***. Please, send your request for the detailed guidelines and forms to the Technical Programme Coordinator by mail, telefax or electronic mail. These forms together with the ***Second Announcement*** contain much useful information. They will be posted to you as quickly as possible.

The Congress Director
Karl Kraus

Note:

If you have access to the **W**orld**W**ide**W**eb, you find regularly updated information about the Congress at the address:

http://www.ipf.tuwien.ac.at/isprs.html

⇨ **Request forms and guidelines from:**

Peter Waldhaeusl
Technical Programme Coordinator
Vienna University of Technology
Gusshausstrasse 27-29 / 122
A-1040 Vienna, Austria

Tel: +43-1-58801 / 3814
Fax:+43-1-505 6268
Email: isprs96@email.tuwien.ac.at

ISSN 0557-1069
KYRIIRI OY 1995