Seventh International Conference on Parallel and Distributed Computing and Systems, Oct. 19 – 21, 1995, Washington D.C. pp.477–480

Design and Implementation of a Multiple-Instruction-Stream Multiple-Execution-Pipeline Architecture

Yamin Li and Wanming Chu

Computer Architecture Laboratory
The University of Aizu, 965-80 Japan

Abstract

This paper describes a single chip Multiple-Instruction-Stream Multiple-Execution-Pipeline (MIS-MEP) architecture capable of improving processor throughput. The MIS-MEP architecture uses multiple instruction dispatch/branch units (slots) to dispatch instructions from multiple instruction streams. Multiple dedicated execution pipelines are provided that are shared by these slots and instructions can be executed on the pipelines in parallel. The slot can communicate and synchronize with one another via queues and synchronization registers. Two examples show how to use the queues and synchronization registers. The MIS-MEP processor can issue 3.945 and 1.633 instructions per cycle in the two examples respectively.

1 Introduction

The processor performance can be measured by the execution time on some suitable work load. The execution time of a program can be expressed as the product of three terms [1]:

$$time = inst\,count * cycles\,per\,inst * time\,per\,cycle$$

In order to maximize performance, a number of different approaches have been used to decrease this quantity. CISC machines attempt to reduce the number of instructions required to complete a task. Most current single-chip RISC processors try to minimize the number of cycles per instruction (CPI) at the expense of an increase in the number of instructions required. To minimize the time per cycle rely primarily on technological advances that involve finding new materials and techniques to make gates that switch faster. In this paper, we introduce a multiple-instruction-stream multiple-execution-pipeline (MIS-MEP) processor architecture that can exploit the instruction level parallelism (ILP) available in a task so that the CPI term can be reduced dramatically.

Several architectures have been developed for exploiting the ILP but the most commercially successful architecture is the superscalar architectures [2], which execute multiple independent instructions in parallel. The superscalar processors issue and execute instructions from a single instruction stream. Several studies have implied that the amount of instruction level parallelism available is between two or three instructions. For this reason, the most superscalar processors can not fully use the multiple dedicated execution pipelines which are responsible for executing corresponding instructions. For example, the MC88110 superscalar microprocessor can issue only at most two instructions per cycle although a set of ten different execution units are provided [3].

The MIS-MEP architecture tries to execute multiple instructions from multiple instruction streams. Multiple instruction dispatch/branch units (slots) fetch instructions from multiple instruction streams and issue them to multiple dedicated execution pipelines. We can consider that each slot makes up a logical processor, a physical MIS-MEP processor consists of multiple logical processors, and the multiple dedicated execution pipelines are shared by these logical processors.

The paper is organized as follows. Section 2 presents the MIS-MEP architecture. Section 3 analyzes the performance potential of MIS-MEP architecture. The final section concludes the paper.

2 The MIS-MEP Architecture

2.1 The MIS-MEP Organization

In the MIS-MEP architecture, multiple slots, multiple dedicated execution pipelines, and multiple register files/queues are provided for executing multiple instructions from multiple instruction streams in parallel. All the slots dispatch instructions on every clock cycle. Instructions are scheduled and issued to multiple dedicated execution pipelines for execution and results are written to register files/queues.

Figure 1 shows the MIS-MEP processor architecture. Differing from a conventional pipelined processor, the MIS-MEP architecture described in this paper consists of four instruction dispatch/branch units (slots). Each slot has its own program counter, status register, and register file/queue. A separate instruction cache is provided for each of slots. On every clock cycle, up to four instructions can be dispatched. Five execution pipelines (two ALU, a floating point adder, a floating point multiplier, and a load/store unit) are provided for executing corresponding instructions. The total 128 general registers are provided for the four slots, each slot has 32 general registers. The 16 synchronization registers can be accessed (read and write) by slots. The queues are used for data transmission between the four slots. The MIS-MEP

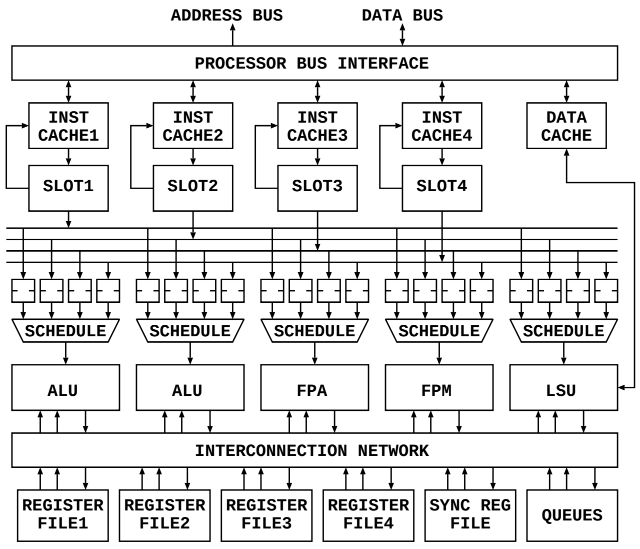

Figure 1: The MIS-MEP processor architecture

processor instruction can fetch source operands from queues and can also write the results to queues in addition to the general registers (see Figure 2).

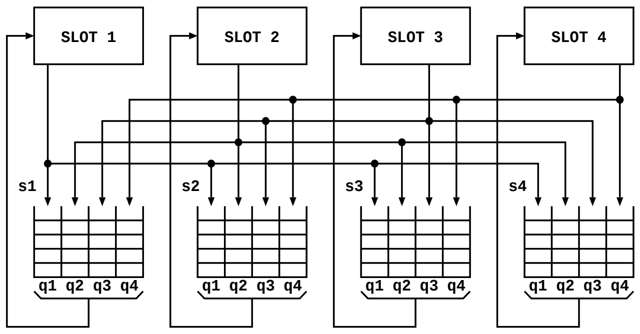

Figure 2: The MIS-MEP processor's queue concept

Dispatched instructions are scheduled by *instruction scheduling units* (ISUs). Each execution pipeline has a ISU. If there is neither resource conflict among four instructions (for example, at most one instruction arrived at ISU of an execution pipeline), nor instruction dependencies with previously issued instructions within a instruction stream, instructions are issued to execution pipelines. The resource conflicts occur when two or more instructions require the same class execution pipelines while the amount of this class execution pipelines is less than the number of requirements.

The ISU selects one instruction to issue to the execution pipeline if dispatched instructions cause resource conflicts. A simple instruction scheduling strategy, *round robin*, is employed. A prioritized method may be used in our architecture at the cost of extra gates and the necessity of priority assignment by user [4].

Availability of source register operands is checked by using the *scoreboard* mechanism. If the scoreboard bits of the source register operands are cleared, a ready instruction is found. Then the source operands are read out from the correct register file and destination register's scoreboard bit is set. The scoreboard bit will be cleared at the final clock cycle of execution stage. Thus the scoreboard bits could prevent incorrect data from entering into the pipeline. The source operands can also be read from queues. The *Empty/Full* mechanism is used for queue access.

The ISU is provided with FIFO registers. Un-issued instructions will be held in the FIFO, waiting for scheduling in the following clock cycle. The corresponding slot is informed to stop fetching instruction in the following clock cycle. Because the next instruction to the un-issued instruction is being fetched because of pipelined operation, the depth of the FIFO must be at least two. The total number of FIFO registers is 2 * 4 * 5, where 4 is the number of slots and 5 is the number of execution pipelines. The execution pipelines carry out the desired data operations, and the results are written back into register file/queues.

An *Un-Blocking Interconnection Network* (UBIN) is needed between the register files/queues and the execution pipelines. From the programmer's point of view, this physical MIS-MEP processor is equal to four logical processors.

2.2 The Pipeline Stages

In our architecture model, each instruction pipeline comprises 4 stages: instruction fetch (IF), schedule and decode (SD), execution (EX), and write back (WB). During the IF-stage, each slot may fetch one instruction from dedicated cache. The instruction fetching unit may receive a "`freeze_fetching`" signal from ISUs. In this case, the fetching operation will be frozen. Resource conflict and source operand availability will be checked in the SD-stage. As mentioned above, the processor use the round robin strategy to schedule the instructions. If an instruction is not issued to execution pipeline, it will be held in the two-word depth FIFO, and, the "`freeze_fetching`" signal will be sent to the corresponding instruction fetching unit. Operands of selected instruction are read in SD-stage. The UBIN provides un-blocking paths for transferring data from the register files/queues to the execution pipelines. The desired data operations are performed in the EX-stage. For most instructions, the execution stage takes one clock cycle, but others take more. The results are written back into register file/queues in the WR-stage. The UBIN also provides un-blocking paths for transferring data from the result-registers to the register files/queues.

2.3 The Instruction Format

All instructions in MIS-MEP processor are 32 bits in length, and are of four formats (see Figure 3). The format 0 instructions have two source operand specifiers and a destination specifier. The format 1 instructions have a source operand specifier, a 12 bit signed constant, and a destination specifier. The format 2 instructions are `bsr` (branch to subroutine) instruction and `bcnd` (conditional branch) instruction, in which the displacement is a 24-bit signed wordoffset. The format 3 instruction is `sethi` (Set High Immediate) instruction in which the immediate is a 24-bit constant that will be used to replace the high-order 24 bits of destination register, the low-order 8 bits will be cleared.

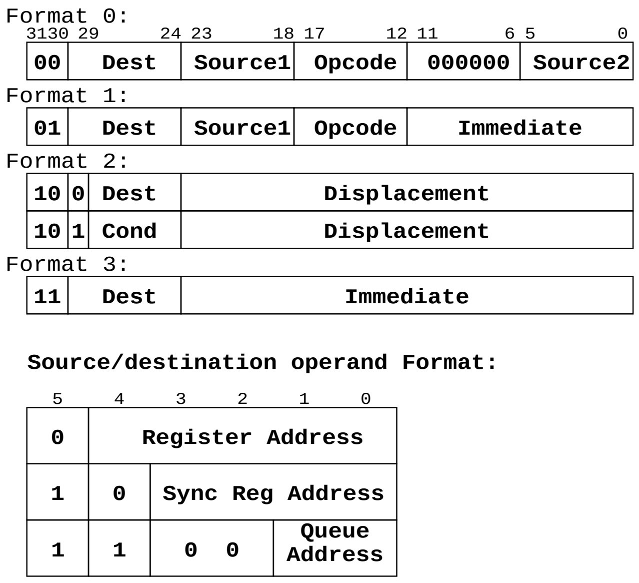

Figure 3: The instruction format of the MIS-MEP processor

As can be seen in Figure 3, a MIS-MEP processor instruction uses 6 bits to specify a source or destination operand. If the first bit is a 0, then the other 5 bits are used to address one of the general registers in the register file. If the first bit is a 1 and second bit is a 0, then the other 4 bits are used to address one of the synchronization registers which are used for synchronization between four instruction streams. If the first and second bits are both 1, then the last 2 bits are used to address one of the queues (refer to Figure 2).

In the MIS-MEP processor architecture, we extend basic RISC instruction set to support the initialization of a new instruction stream and the synchronization of two instruction streams with following instructions.

```
sfork - to create a new instruction stream,
sjoin - to terminate an instruction stream,
sbclr - synchronization bit clear, and
sbset - synchronization bit set.
```

The `sfork` and `sjoin` instructions are used to create and terminate an instruction stream. The `sbclr` and `sbset` are used to synchronize two instruction streams. Each bit of a synchronization register can be used for this purpose.

3 Analysis

In order to analyze the performance potential of the MIS-MEP architecture, several small benchmarks were hand compiled and optimized. Two examples have been chosen to show how to use the synchronization registers and queues and to illustrate the abilities of the MIS-MEP architecture.

The first example is the Lawrence Livermore Loops. We use the C source version of LLL #3 for purposes of this analysis [5]:

```
main()
{
  float z[1000], x[1000], q;
  int k;
  q=0.0;
  for(k=0;k<1000;k++)
    q+=z[k]*x[k];
}
```

The assemble codes for general-purpose RISC processor are shown as following.

```
add   r2,r0,1000   ; k=1000, [r0] is zero.
add   r7,r0,r0     ; q=0.0
sethi r3,high(z)   ; z high 24 bits base address
sethi r4,high(x)   ; x high 24 bits base address
sethi r5,high(q)   ; q high 24 bits address
or    r3,r3,low(z) ; z low 8 bits base address
or    r4,r4,low(x) ; x low 8 bits base address
or    r5,r5,low(q) ; q low 8 bits address
@L100:
ld    r8,r3,0      ; load z[k]
ld    r9,r4,0      ; load x[k]
add   r3,r3,4      ; update index of z
fmul  r8,r9,r8     ; z[k]*x[k]
subcc r2,r2,1      ; k=k-1 and set condition code
add   r4,r4,4      ; update index of x
bgt   @L100        ; if k>0,continue, delay branch
fadd  r7,r8,r7     ; q=q+z[k]*x[k], delay slot
st    r5,r7        ; store q
```

We use the extended instructions to create three new instruction streams. Together with original stream, there are four instruction streams and each performs one of fourth loop operations. In the loop bodies, four instruction streams were executed in fully parallel as shown in Figure 4. Note that the `bgt` was executed in the Instruction Dispatch/Branch Unit. Consider the total performance (taking account of the other parts of instructions), we got a CPI rate of 0.253, i.e., 3.945 instructions were allowed to issue each clock cycle.

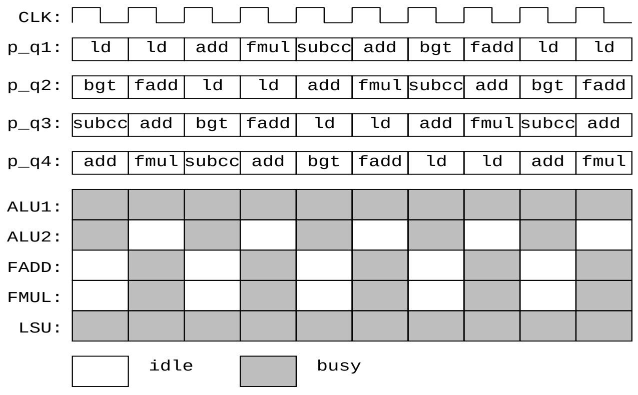
CLK:
p_q1: ld ld add fmul subcc add bgt fadd ld ld
p_q2: bgt fadd ld ld add fmul subcc add bgt fadd
p_q3: subcc add bgt fadd ld ld add fmul subcc add
p_q4: add fmul subcc add bgt fadd ld ld add fmul
ALU1:
ALU2:
FADD:
FMUL:
LSU:
idle
busy

Figure 4: The execution of loop body of example 1

The second example program is `strcpy()` function. Compare to the first example, this function has little parallelism due to the dependency found in the loop control variable.

```
main()
{
  char *str1="This is an example string", str2[26];
  while (*str2++=*str1++);
}
```

The assemble codes for general-purpose RISC processor are shown as following.

```
sethi r2,high(str1)   ; str1 high 24 bits address
sethi r3,high(str2)   ; str2 high 24 bits address
or    r2,r2,low(str1) ; str1 low 8 bits address
or    r3,r3,low(str2) ; str2 low 8 bits address
@L100:
ldb   r4,r2,0         ; load a character of str1
add   r2,r2,1         ; update index of str1
subcc r0,r4,r0        ; determine if end
stb   r3,r4           ; store a character of str2
bgt   @L100           ; if not, continue, delay br.
add   r3,r3,1         ; update index of str2
```

Four slots work in following manner. The slot 4 performs destination address calculation. The calculated address is put into queue4 of slot 1 for store operation. The slot 3 performs source address calculation and the calculated address is put into queue3 of slot 2 for load operation. The slot 2 is responsible for loading characters of source string according to the addresses found in the queue3 and put the characters into queue2 of slot 1. The slot 1 is responsible for loop control and store character store. It gets the characters from queue2, and gets addresses from queue4 for store operation. In the following code segments, we use `s1, s2, s3`, and `s4` to refer to destination queues and use `q1, q2, q3`, and `q4` to refer to source queues.

```
---------------------------------------------------
SLOT 1:               ; loop control and char store
---------------------------------------------------
sfork slot2,r2        ; create other three streams
sfork slot3,r3        ; r2, r3, and r4 contain the
sfork slot4,r4        ; start addresses of streams
subcc r2,q2,r0        ; get char from q2, end?
@L100:
stb   q4,r2           ; store char, q4: mem-adr
bnz   @L100           ; if not, continue, delay br.
subcc r2,q2,r0        ; get char from q2, end?
sjoin slot2           ; terminate the inst stream 2
sjoin slot3           ; terminate the inst stream 3
sjoin slot4           ; terminate the inst stream 4
---------------------------------------------------
SLOT 2:               ; load characters
---------------------------------------------------
@L100:
ldb   s1,q3           ; load char to s1, q3:mem-adr
bra   @L100           ; branch always
nop                   ; no operation
---------------------------------------------------
SLOT 3:               ; source address calculation
---------------------------------------------------
sethi r2,high(str1)   ; str1 high 24 bits address
or    r2,r2,low(str1) ; str1 low 8 bits address
@L100:
mov   s2,r2           ; send adr to s2 for load
bra   @L100           ; branch always
add   r2,r2,1         ; update adr of str1 (source)
---------------------------------------------------
SLOT 4:               ; dest address calculation
---------------------------------------------------
sethi r2,high(str2)   ; str2 high 24 bits address
or    r2,r2,low(str2) ; str2 low 8 bits address
@L100:
mov   s1,r2           ; send adr to s1 for store
bra   @L100           ; branch always
add   r2,r2,1         ; update adr of str2 (dest)
---------------------------------------------------
```

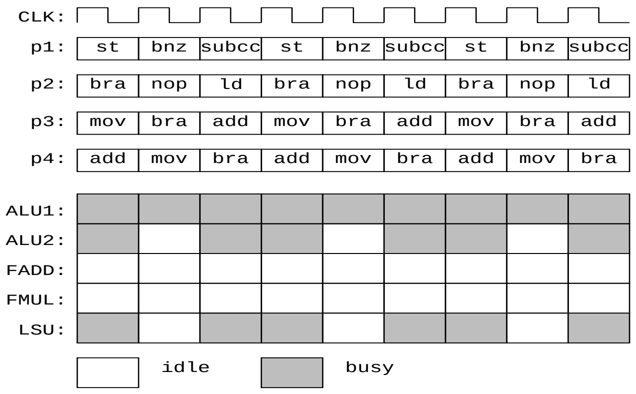
CLK:
p1: st bnz subcc st bnz subcc st bnz subcc
p2: bra nop ld bra nop ld bra nop ld
p3: mov bra add mov bra add mov bra add
p4: add mov bra add mov bra add mov bra
ALU1:
ALU2:
FADD:
FMUL:
LSU:
idle
busy

Figure 5: The execution of loop body of example 2

The four instruction streams' execution sequences are shown in Figure 5. We got a total CPI rate of 0.613, i,e, 1.633 instructions were allowed to issue each clock cycle. Achieving a CPI rate of about 0.6 for this example is an excellent result.

4 Conclusion Remarks

In this paper, we have described the MIS-MEP architecture designed to exploit instruction level parallelism. The MIS-MEP processor features multiple instruction dispatch/branch units sharing multiple dedicated execution pipelines and communicating and synchronizing with one another via queues and synchronization registers. A number of new instructions have been developed for the multiple instruction streams' management. Two examples, one holds sufficient parallelism and the other displays little parallelism, have been selected for showing the performance potential of the MIS-MEP architecture.

References

[1] J. hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990

[2] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1990.

[3] K. Diefendorff and M. Allen,“Organization of the Motorola 88110 Superscalar RISC Microprocessor,” in *IEEE MICRO*, April 1992.

[4] S. Fiske and W. dally, “Thread Prioritization: A Thread Scheduling Mechanism for Multiple-Context Parallel Processors. in *Proc. of the First IEEE Symposium on High-Performance Computer Architecture*, January 1995.

[5] G. Tyson, M. farrens, and A. Pleszkun, “MISC: A Multiple Instruction Stream Computer”, in *Proc. of the 25th annual International Symposium on Microarchitecture*, December 1992.