MIPS on FLEET

Amir Kamil
Computer Science Division, University of California, Berkeley
kamil@cs.berkeley.edu

December 18, 2006

1 Introduction

The introduction of multi-core processors by the major microprocessor developers marks a dramatic shift in software development: parallelism for laptops and desktop machines will no longer be hidden within a micro-architecture, but will be exposed to higher level software. Programmer control of concurrency and data movement will be necessary in order to achieve maximum performance. Unfortunately, most modern architectures and programming techniques were originally designed for sequential programming, and it is not clear that they are ideal in a parallel setting. The FLEET computer [1] is a new architecture designed from the ground up with parallelism in mind, providing the programmer with direct control over data movement and concurrency.

Since FLEET is a completely new architecture, programming for it can be quite challenging. The machine is inherently parallel, so sequential programming techniques do not directly translate to it. In order to alleviate this problem, we have developed a translator between MIPS binary code and FLEET assembly. Using this translator, programs written in MIPS assembly can run on FLEET. The eventual goal is to allow code written in higher-level languages to also run on FLEET by first compiling to MIPS assembly. In this paper, we present the techniques used by the translator to convert MIPS code to FLEET code.

2 Background

2.1 MIPS Assembly

We chose MIPS R3000 assembly as the source of the translator for many reasons. MIPS is a RISC architecture, so the instructions it provides are simple and highly regular. In addition, it is widely used in embedded systems, so it has a large code base and many tools, such as the SPIM emulator [3].

The translator currently supports the 32 numbered MIPS registers, as well as the `$hi` and `$lo` registers used by multiplication and division. It implements integer arithmetic and logic, branches, jumps, sets, loads, stores, data movement between `$hi` and `$lo` and the numbered registers, and basic system

calls. The translator does not simulate the MIPS instruction pipeline, and does not insert any delay slots into the source code.

2.2 SHIPs

The FLEET architecture consists of many small processing elements, called *SHIPs*. Each element can have input and output ports, which are connected by a *switch fabric* to every other port in the machine. Computation is performed exclusively through data movement between these ports. As such, FLEET can be considered a *one-instruction* machine, since the only instructions it provides are variants of a single data movement command, as specified in AM10 and AM11 [4, 6].

We used the SHIPs defined in IES31 [5] as the basis of the translator, with some differences. The Fetch SHIP was modified from IES31 to allow a codebag to be released or revoked. The input and output ports were replaced with the boxes of AM10 and AM11. In addition, we found it useful to define new SHIPs in order to simplify the task of the translator.

3 Implementation

In this section, we discuss the details of how MIPS assembly code is translated into FLEET assembly.

3.1 Instruction Framework

The translator implements each MIPS instruction as a separate codebag. This ensures that instructions can be properly sequenced so that they do not interfere with each other. Each instruction codebag dispatches the next one to the Fetch SHIP, and only releases it when all side effects of the instruction have completed.

3.2 Registers

We implemented MIPS registers on top of the FIFO storage SHIP defined in IES31. The translator only uses a single element of each FIFO to represent a register.

Registers can never be empty, so they must be initialized by inserting a zero into their respective FIFO units. A register read simply copies the output of the FIFO, while a read must destroy the old output and insert the new input. The translator uses register writes for sequencing, so the standing moves on their inputs are torn down. The translator generally uses a `accept+ack` at the input of each FIFO, with the acknowledgment releasing the next instruction codebag to execute.

3.3 Arithmetic Operations

Addition and subtraction instructions are implemented using the Adder SHIP of IES31. Since IES31 does not define a SHIP for multiplication or division, we constructed a new MultiplyDivide SHIP with two outputs. This SHIP has the ability to perform both signed and unsigned multiplication and division. For multiplication, the outputs are used for the upper and lower bits of the result, while for division, they are used for the quotient and the remainder.

The output of the MultiplyDivide SHIP correspond exactly to what should be placed in the `$hi` and `$lo` registers upon executing multiplication and division in MIPS. This has a slight complication for sequencing, since the next instruction codebag cannot be released until the writes to both registers have completed. To accomplish this, we defined a TokenCombine SHIP that converts two tokens into one. The write notifications are sent to this SHIP, and its output is used to release the next codebag.

At the current moment, the translator does not check for arithmetic overflow in any of these instructions.

3.4 Logical Operations

The Bitwise SHIP of IES31 allows any of 256 bitwise operations to be performed on the inputs. This is far more than necessary to support MIPS, and since no one has yet implemented this SHIP, we defined a new Logic SHIP with support for the basic logical operations, such as `and`, `or`, `xor`, and `nor`.

3.5 Shift Operations

MIPS assembly allows left, right arithmetic (i.e. with sign extension), and right logical (i.e. without sign extension) shifts. The shift amount can be specified statically in the actual instruction, or dynamically in a register.

IES31 defines a Shift SHIP that can perform various shifts of a single bit. This SHIP, however, appears unsuitable for use with the MIPS instructions, since each would translate into a large number of FLEET instructions, resulting in a lot of data movement through the switch fabric. The register-specified shifts in particular would be quite complicated, since they would have to be implemented using a dynamic loop. In order to keep things simple, we defined a new version of the Shift SHIP that allows for variable shifts.

3.6 Branches and Sets

The branch instructions in MIPS transfer of control to a nearby location, given a certain condition. The target can always be computed statically. The set instructions write either zero or one to a target register, given a certain condition. Both operations perform comparisons and then select between two values, in the former case between two target instruction codebags, and in the latter, between zero and one.

The Adder SHIP provides a command for selection that can be used to implement branches and sets. It also has rudimentary support for comparisons. However, for simplicity and since a dedicated comparator would be faster than using the Adder, we implemented a new Comparator SHIP to perform the comparisons.

3.7 Jumps

MIPS provides both jumps to absolute targets and to an address contained in a register. The former can be implemented by just dispatching the target instruction codebag. The latter, however, proves problematic, since it relies on the relationship between instruction addresses and codebags. The FLEET interpreter translates codebag names statically into integers in some undefined manner, which it then uses at runtime.

Since the name translation is undefined, the MIPS translator cannot perform the name translation on its own.

Our solution is to define a new MIPSAddressFetch SHIP that, given a MIPS address, dispatches the associated codebag. This SHIP relies on hooks into the interpreter, so it would need to redefined if another interpreter or simulator were used.

3.8 Memory Access Operations

Memory in MIPS is byte-addressed, and instructions are provided for loading and storing bytes, half-words, and words. On the other hand, the MemoryRead and MemoryWrite SHIPs in IES31 only provide word-addressed memory[1] Not only would it be difficult to implement the sub-word instructions in this scheme, but every MIPS address would have to be divided by the word length (in bytes) before it could be sent to one of the SHIPs. This adds extra computation and data movement to each memory accesses.

In order to avoid the complexity and cost of using the word-addressed SHIPs, we implemented new byte-addressed ByteMemoryRead and ByteMemoryWrite SHIPs. The actual input size is provided to these SHIPs, so they can natively handle bytes, half-words, and words. In addition, the ByteMemoryRead SHIP optionally provides sign extension for sub-word reads.

3.9 Data Movement

Since MIPS places the results of a multiplication or division in the `$hi` and `$lo` registers, it provides instructions for moving data between them and the numbered registers. These instructions are implemented directly using the read and write mechanisms of §3.2.

3.10 System Calls

MIPS provides a `syscall` instruction for performing system calls provided by the operating system. The SPIM simulator provides a number of such calls. We implemented the SPIM calls for printing an integer, reading an integer, printing a string, and exiting the program.

Since FLEET does not yet have an input/output specification, we implemented a basic MIPSSyscall SHIP that can be used with the FLEET interpreter to perform the above calls. The printing and reading calls use the the standard input and output streams of the interpreter. The exit call is implemented by dispatching to a cleanup and halt codebag. As in §3.7, this done using hooks into the interpreter.

3.11 Cleanup

At the end of the translated MIPS instruction stream, the FLEET machine must be restored to pristine condition. A cleanup codebag does so by restoring the standing moves on register inputs and emptying them, as well as any other cleanup required.

[1] IES31 does not actually specify a width for words. It is assumed, however, that this would be at least 32 bits.

4 Results

The implementation currently supports the entire functionality discussed in §3. We have verified operation of each supported MIPS instruction using unit testing. In addition, we have run a simple program to compute the Fibonacci numbers [2] through the translator. The program reads a number n from the console and outputs the nth Fibonacci number. We loaded the assembly code into SPIM, used the `dump` command to convert it into binary format, and ran the result through the translator. We then verified its operation by running the resulting FLEET code through the FLEET interpreter. The MIPS and FLEET assembly code for a portion of the program are provided in §A.

5 Discussion

While a significant amount of MIPS assembly is implemented by the translator, there are a few unimplemented instructions, and minor bugs in some of the implemented instructions. In order to better support the translator, the FLEET interpreter had to be modified slightly. Finally, the translation to FLEET assembly is quite naïve, with no optimization to take advantage of the concurrency.

5.1 Bugs and Limitations

At the moment, the translator only supports a handful of system calls, does not implement exception handling, and does not support any coprocessor operations, including floating point instructions. It also can only process binaries composed solely of a text segment, such as the output of the `dump` command in SPIM. As of now, the translator is purely sequential: each MIPS instruction must complete before the next one is started.

The register jump instructions currently rely on hooks into the FLEET interpreter to work, as discussed in §3.7. A specification for mapping codebag names to integer identifiers is necessary to remove this hack.

Since the translation is done statically, self-modifying code will not work with it.

In order to run arbitrary programs on FLEET, an input/output specification needs to be defined. FLEET also needs a floating point specification before floating point instructions can be implemented.

5.2 Modifications to the FLEET Interpreter

The FLEET interpreter had to be modified slightly to support certain MIPS instructions.

5.2.1 Register Jumps

In order to support register jumps, the interpreter has to include a mechanism for converting codebag names to integer identifiers. The only modification necessary was to expose the root codebag to the world, since the CodeBag class already had an interface for doing the conversion.

5.2.2 Byte-Addressed Memory Access

In order to support the sub-word memory access operations in MIPS, FLEET must provide byte-addressed memory. The interpreter was modified to support little-endian byte-addressed reading and writing.

5.2.3 I/O

In order to run basic programs using the interpreter, it is necessary to provide at least a minimal level of input and output. The interpreter was modified to allow input and output files to the interpreter to be specified, thereby freeing standard input and output for use by the actual code.

5.2.4 Suggested Modifications

Other interpreter changes would also be useful to the MIPS translator and FLEET programs in general.

In MIPS code, the standard initial stack location is at the address `0x7FFFFFFF`. The FLEET interpreter, however, is naïve in its memory implementation, using only a flat array. Using the standard stack location would therefore result in a very large array for memory in the interpreter, so a non-standard address must be used. It would be beneficial to modify the interpreter to represent memory in a more intelligent manner.

A program may wish to use both word-addressed and byte-addressed memory. Instead of using different SHIPs for each, it would be better to combine the two into a single SHIP. This would require adding an extra input to the resulting SHIP, specifying which mode to use. To make this change seamless to existing code would require initializing the SHIP with a standing copy on this input, specifying word-addressing. As such, it would be useful to allow SHIP-specific initialization of inboxes and outboxes.

5.3 Future Work

The MIPS translator currently enforces strict sequencing between MIPS instructions, wasting much of the concurrency present in FLEET. The translator should be able to relax the sequencing where possible, or even implement instruction-level parallelism (ILP) to take advantage of duplicated SHIPS or multiple FLEETs. It should be possible to leverage the ILP work done for modern architectures to do this.

6 Conclusion

As multi-core processors enter the mainstream, managing concurrency and data movement will be two of the major tasks in producing highly optimized code. The FLEET architecture has the potential to revolutionize concurrent programming, since it places direct control of concurrency and data movement in the hands of the programmer.

However, with FLEET's emphasis on concurrency, sequential programming is not trivial on it. No compilers exist that target FLEET, so very few programs exist that will run on FLEET. Our work in this paper alleviates both problems, allowing existing, sequential MIPS programs to run on FLEET, and eventually allowing compilers to target FLEET by using MIPS as an intermediary. We believe that this

work is the first step towards being able to run arbitrary sequential, as well as concurrent, programs on FLEET.

References

[1] The FLEET project. Website: http://research.cs.berkeley.edu/class/fleet.

[2] MIPS Fibonacci program. Taken from http://www.cs.uic.edu/~i366/notes/SPIM%20Examples.html and modified to remove the data segment.

[3] J. Larus. SPIM: A MIPS32 simulator. Website: http://www.cs.wisc.edu/~larus/spim.html.

[4] A. Megacz and I. Sutherland. Boxes. http://research.cs.berkeley.edu/class/fleet/docs/people/adam.megacz/am10-Boxes.pdf.

[5] A. Megacz and I. Sutherland. Some SHIPs. http://research.cs.berkeley.edu/class/fleet/docs/people/ivan.e.sutherland/ies31-Some.SHIPs.pdf.

[6] The Students and Instructors of CS-294-11, edited by A. Megacz. Boxes: Data, token, in, out. http://research.cs.berkeley.edu/class/fleet/docs/people/adam.megacz/am11-Boxes:.Data,Token,In,Out.pdf.

A Fibonacci Program

In this section, we provide the code for the main recursive piece of the Fibonacci program. For brevity, we leave out the base case tests.

A.1 MIPS Code

The following is the MIPS assembly code for the recursive section:

```
recurse:
  addi  $t0,$t0,-1      # $t0 = n-1
  sw    $t0,($sp)       # push argument n-1 on stack
  addi  $sp,$sp,-4      #   and decrement stack pointer
  jal   Fib             # call Fibonacci with argument n-1
                        # leave result on stack for now
  lw    $t0,($fp)       # re-copy argument to $t0:  $t0 = n
  addi  $t0,$t0,-2      # $t0 = n-2
  sw    $t0,($sp)       # push argument n-2 on stack
  addi  $sp,$sp,-4      #   and decrement stack pointer
  jal   Fib             # call Fibonacci with argument n-2
  addi  $sp,$sp,4       # increment stack pointer
  lw    $t0,($sp)       #   and pop result of Fib(n-2) from into $t0
  addi  $sp,$sp,4       # increment stack pointer
  lw    $t1,($sp)       #   and pop result of Fib(n-1) from into $t1
  add   $t0,$t0,$t1     # $t0 = Fib(n-2) + Fib(n-1); have result
```

A.2 FLEET Code

The following is the recursive section of the FLEET code output by the translator:

```
PC4000d8: {
  // addi $t0, $t0, -1
  copy t0Fifo.out -> adder.A
  -1 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  PC4000dc -> fetch.codebag
  discard fetch.done -> ()
  discard t0Fifo.out -> ()
  move adder.out -> t0Fifo.in
  accept+ack t0Fifo.in -> fetch.release
}
PC4000dc: {
  // sw $t0, 0($sp)
  copy spFifo.out -> adder.A
  0 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  move adder.out -> memwrite.addr
  0 -> memwrite.stride
  1 -> memwrite.count
  4 -> memwrite.size
  PC4000e0 -> fetch.codebag
  discard fetch.done -> ()
  copy t0Fifo.out -> memwrite.data
  move memwrite.done -> fetch.release
}
PC4000e0: {
  // addi $sp, $sp, -4
  copy spFifo.out -> adder.A
  -4 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  PC4000e4 -> fetch.codebag
  discard fetch.done -> ()
  discard spFifo.out -> ()
  move adder.out -> spFifo.in
  accept+ack spFifo.in -> fetch.release
}
PC4000e4: {
```

```
  // jal 1048619
  PC4000ac -> fetch.codebag
  discard raFifo.out -> ()
  4194536 -> raFifo.in
  accept+ack raFifo.in -> fetch.release
  discard fetch.done -> ()
}
PC4000e8: {
  // lw $t0, 0($fp)
  copy fpFifo.out -> adder.A
  0 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  move adder.out -> memread.addr
  0 -> memread.stride
  1 -> memread.count
  4 -> memread.size
  1 -> memread.signed
  PC4000ec -> fetch.codebag
  discard fetch.done -> ()
  discard t0Fifo.out -> ()
  move memread.data -> t0Fifo.in
  accept+ack t0Fifo.in -> fetch.release
  discard memread.done -> ()
}
PC4000ec: {
  // addi $t0, $t0, -2
  copy t0Fifo.out -> adder.A
  -2 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  PC4000f0 -> fetch.codebag
  discard fetch.done -> ()
  discard t0Fifo.out -> ()
  move adder.out -> t0Fifo.in
  accept+ack t0Fifo.in -> fetch.release
}
PC4000f0: {
  // sw $t0, 0($sp)
  copy spFifo.out -> adder.A
  0 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
```

```
  move adder.out -> memwrite.addr
  0 -> memwrite.stride
  1 -> memwrite.count
  4 -> memwrite.size
  PC4000f4 -> fetch.codebag
  discard fetch.done -> ()
  copy t0Fifo.out -> memwrite.data
  move memwrite.done -> fetch.release
}
PC4000f4: {
  // addi $sp, $sp, -4
  copy spFifo.out -> adder.A
  -4 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  PC4000f8 -> fetch.codebag
  discard fetch.done -> ()
  discard spFifo.out -> ()
  move adder.out -> spFifo.in
  accept+ack spFifo.in -> fetch.release
}
PC4000f8: {
  // jal 1048619
  PC4000ac -> fetch.codebag
  discard raFifo.out -> ()
  4194556 -> raFifo.in
  accept+ack raFifo.in -> fetch.release
  discard fetch.done -> ()
}
PC4000fc: {
  // addi $sp, $sp, 4
  copy spFifo.out -> adder.A
  4 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  PC400100 -> fetch.codebag
  discard fetch.done -> ()
  discard spFifo.out -> ()
  move adder.out -> spFifo.in
  accept+ack spFifo.in -> fetch.release
}
PC400100: {
  // lw $t0, 0($sp)
```

```
  copy spFifo.out -> adder.A
  0 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  move adder.out -> memread.addr
  0 -> memread.stride
  1 -> memread.count
  4 -> memread.size
  1 -> memread.signed
  PC400104 -> fetch.codebag
  discard fetch.done -> ()
  discard t0Fifo.out -> ()
  move memread.data -> t0Fifo.in
  accept+ack t0Fifo.in -> fetch.release
  discard memread.done -> ()
}
PC400104: {
  // addi $sp, $sp, 4
  copy spFifo.out -> adder.A
  4 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  PC400108 -> fetch.codebag
  discard fetch.done -> ()
  discard spFifo.out -> ()
  move adder.out -> spFifo.in
  accept+ack spFifo.in -> fetch.release
}
PC400108: {
  // lw $t1, 0($sp)
  copy spFifo.out -> adder.A
  0 -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  move adder.out -> memread.addr
  0 -> memread.stride
  1 -> memread.count
  4 -> memread.size
  1 -> memread.signed
  PC40010c -> fetch.codebag
  discard fetch.done -> ()
  discard t1Fifo.out -> ()
  move memread.data -> t1Fifo.in
```

```
  accept+ack t1Fifo.in -> fetch.release
  discard memread.done -> ()
}
PC40010c: {
  // add $t0, $t0, $t1
  copy t0Fifo.out -> adder.A
  copy t1Fifo.out -> adder.B
  "ADD NONE" -> adder.cmd
  0 -> adder.linkin
  PC400110 -> fetch.codebag
  discard fetch.done -> ()
  discard t0Fifo.out -> ()
  move adder.out -> t0Fifo.in
  accept+ack t0Fifo.in -> fetch.release
}
```