ForwardCom: An open-standard instruction set for high-performance microprocessors

Agner Fog

August 1, 2016

Contents

Chapter 1

Introduction

ForwardCom stands for Forward Compatible Computer system.

This document proposes a new open instruction set architecture designed for optimal performance, flexibility and scalability. The ForwardCom project includes both a new instruction set architecture and the corresponding ecosystem of software standards, application binary interface (ABI), memory management, development tools, library formats and system functions. This project illustrates the improvements that can be obtained by a complete vertical redesign of hardware and software based on an open, collaborative process.

A short introduction to ForwardCom is provided at http://www.forwardcom.info.

This manual and all associated code is maintained at https://github.com/ForwardCom.

1.1 Highlights

- The ForwardCom instruction set is a compromise between the RISC and CISC principles, combining the fast and streamlined decoding and pipeline design of RISC systems with the compactness and more work-done-per-instruction of CISC systems.

- The ForwardCom design is scalable to support small embedded systems as well as large supercomputers and vector processors without losing binary compatibility.

- Vector registers of variable length are provided for efficient handling of large data sets.

- Array loops are implemented in a new flexible way that automatically uses the maximum vector length supported by the microprocessor in all but the last iteration of a loop. The last iteration automatically uses a vector length that fits the remaining number of elements. No extra code is needed to deal with remaining data and special cases. There is no need to compile the code separately for different microprocessors with different vector lengths.

- No recompilation or update of software is needed when a new microprocessor with longer vector registers becomes available. The software is guaranteed to be forward compatible and take advantage of the longer vectors of new microprocessor models.

- Strong security features are a fundamental part of the hardware and software design.

- Memory management is simpler and more efficient than in traditional systems. Various techniques are used for avoiding memory fragmentation. There is no memory paging and no translation lookaside buffer (TLB). Instead, there is a memory map with a limited number of sections with variable size.

- There are no dynamic link libraries (DLLs) or shared objects. Instead, there is only one type of function libraries that can be used for both static and dynamic linking. Only the part of the library that is actually used is loaded and linked. The library code is kept contiguous with the

3

main program code in almost all cases. It is possible to automatically choose between different versions of a function or library at load time, based on the hardware configuration, operating system, or user interface framework.

- A mechanism for calculating the required stack size is provided. This can prevent stack overflow in most cases without making the stack bigger than necessary.

- A mechanism for optimal register allocation across program modules and function libraries is provided. This makes it possible to keep most variables in registers without spilling to memory. Vector registers can be saved in an efficient way that stores only the part of the register that is actually used.

1.2 Background

An instruction set architecture is a standardized set of machine instructions that a computer can run. There are many instruction set architectures in use.

Some commonly used instruction sets are poorly designed from the beginning. These systems have been augmented many times with extensions and patches. One of the worst cases is the widely used x86 instruction set and its many extensions. The x86 instruction set is the result of a long history of short-sighted extensions and patches. The result of this development history is a very complicated architecture with thousands of different instruction codes, which is very difficult and costly to decode in a microprocessor. We need to learn from past mistakes in order to make better choices when designing a new instruction set architecture and the software that supports it.

The design should be based on an open process. Krste Asanović and David Patterson have presented compelling arguments for why an open instruction set should be preferred. Openness can be crucial for the success of a technical design. For example, the original IBM PC in the early 1980's had an advantage over competing computers because the open architecture allowed other hardware and software producers to make compatible equipment. IBM lost their market dominance when they switched to the proprietary Micro Channel Architecture in 1987. The successes of open source software are well known and need no further discussion here. The only thing that is missing for a complete computer ecosystem based on open standards is an open microprocessor architecture. This will open the market also for smaller microprocessor producers and niche products.

This manual is based on discussions in various Internet forums. The specifications are preliminary. The development of a new standard should benefit from a long experimental phase, and it would be unwise to make it a fixed standard at this initial stage.

1.3 Design goals

Previously published open instruction sets are suitable for small, cheap microprocessors for embedded systems, system-on-a-chip designs, FPGA implementations for scientific experiments, etc. The proposed ForwardCom architecture takes the idea further and aims at a design that can outperform existing high-end processors.

The ForwardCom instruction set architecture is based on the following priorities:

- The instruction set should have a simple and consistent modular design.

- The instruction set should represent a suitable compromise between the RISC principle that enables fast decoding, and the CISC principle that makes it possible to do more work per instruction and to use the code cache more efficiently.

- The design should be extensible so that new instructions and extensions can be added in a consistent and predictable way.

- The design should be scalable so that it is suitable for both small computers with on-chip RAM and large supercomputers with very long vectors.

- The design should be competitive over current commercial designs with a focus on the high-end applications of tomorrow rather than the low-end applications of yesterday.

- Vector support and other features that have proven essential for high performance should be a fundamental part of the design, not a clumsy appendix.

- Security should be a fundamental part of the design, not patches added ad hoc.

- The instruction set should be designed through an open process with the participation of the international hardware and software community, similar to the standardization work in other technical areas.

- The entire vertical design should be non-proprietary and allow anybody to make compatible software, hardware and equipment for test, debugging and emulation.

- Decisions about instructions and extensions should not be determined by the short term marketing considerations of an oligopolistic microprocessor industry but by the long term needs of the entire hardware and software community and organizations.

- The design should allow the construction of forward compatible software that will run optimally without recompilation on future processors with larger vector registers.

- The design should allow application-specific extensions.

- The basic aspects of the ecosystem of ABI standard, assembler, compilers, function libraries, system functions, user interface framework, etc. should also be standardized for maximum compatibility.

A new instruction set will not easily get success on a commercial market, even if it is better than legacy systems, because the market prefers backward compatibility with existing software and hardware. It is unlikely that the ForwardCom instruction set will make a successful commercial product within a short time frame, but the discussion about what an ideal instruction set and software ecosystem might look like is still useful. The ForwardCom project has already generated so many important new ideas that it is worth pursuing further, even if we don't know where this will end. The present work can be useful if the need for introducing a new instruction set architecture should arise for other reasons. It will be particularly useful for large vector processors, for applications where security is important, for real-time operating systems, as well as for projects where the patent and license restrictions of other architectures would be an obstacle.

The proposals in this document may also be useful as a source of inspiration and for scientific experiments. Many of the ideas are independent of the design details and may be implemented in existing systems.

1.4   Comparison with other open instruction sets

A few other open instruction sets have been proposed, most notably RISC-V and OpenRISC. Both are pure RISC designs with mostly fixed 32-bit instruction word sizes. These instruction sets are suitable for small systems where the use of silicon space is economized, but they are not designed for high performance superscalar processors and they do not focus on details that are critical for achieving maximum performance in bigger systems. The present proposal is thought as the next step towards making an open instruction set that is actually more efficient than the best commercial instruction sets today.

A typical RISC design with the instruction size limited to 32 bits leaves only limited space for immediate constants and addresses of memory operands. A medium size program will need 32-bit relative

addresses of static memory operands to avoid overflow during the relocation process in the linker.
A 32-bit relative address requires several instructions in the pure RISC designs. For example, to add
a memory operand to the value of a register, you need five instructions in a RISC design with only
32-bit instruction words: (1) load the lower part of the 32-bit address offset, (2) add the upper part
of the 32-bit address offset, (3) add the reference pointer or instruction pointer to this value, (4)
read the memory operand from the calculated address, (5) do the desired addition. The ForwardCom
design does all this in a single instruction with double word size. The speed advantage is obvious.
The address calculation, load, and execution are done at each their stage in the pipeline in order to
achieve a smooth throughput of one instruction per clock cycle in each pipeline lane.

Another important difference is that the previous RISC designs have limited support for vector oper-
ations. The ForwardCom design introduces a new system of variable-length vector registers that is
more efficient and flexible than the best current commercial designs. Efficient vector operations are
essential for obtaining maximum performance, and this has been an important priority in the design of
the ForwardCom architecture proposed here.

1.5   References and links

- Krste Asanović and David Patterson: "The Case for Open Instruction Sets. Open ISA Would
  Enable Free Competition in Processor Design". Microprocessor Report, August 18, 2014.
  www.linleygroup.com/mpr/article.php?id=11267

- RISC-V: The Free and Open RISC Instruction Set Architecture riscv.org

- OpenRISC: openrisc.io

- Open Cores: opencores.org

- Agner Fog: Proposal for an ideal extensible instruction set, 2015. A blog discussion thread that
  initiated the ForwardCom project.
  www.agner.org/optimize/blog/read.php?i=421

- Agner Fog: Stop the instruction set war, 2009. Blog post about the problems with the x86 in-
  struction set.
  www.agner.org/optimize/blog/read.php?i=25

- Darek Mihocka: Standard Need To Be Forward Looking, 2007. Blog post criticizing the x86
  instruction set standard.
  www.emulators.com/docs/nx02_standards.htm. See also the following pages.

Chapter 2

Basic architecture

This chapter gives an overview of the most important features of the ForwardCom instruction set architecture. Details are given in the subsequent chapters.

2.1 A fully orthogonal instruction set

The ForwardCom instruction set is fully orthogonal in all respects. The same instruction can use integer operands of all sizes and floating point operands of all precisions. It can use register operands, memory operands or immediate operands. It can use many different addressing modes. Instructions can be coded in short forms with two operands where the same register is used for destination and source operand, or longer forms with three operands. It can work with scalars or vectors of any size. It can have predication or masks for conditional execution at the vector element level, and it can have optional flag inputs for deciding rounding mode, exception control and other details, where appropriate. Data constants of all types can be included in the instructions and compressed in various ways to reduce the instruction size.

Rationale

The orthogonality is implemented by a standardized modular design that makes the hardware implementation simpler. It also makes compilation simpler and more flexible and makes it easier for the compiler to convert linear code to vector code.

The support for immediate constants of all types is an improvement over current systems. Most current systems store floating point constants in a data segment and access them through a 32-bit address in the instruction code. This is a waste of data cache space and causes many cache misses because the data are scattered around in different sections. Replacing a 32-bit address with a 32-bit immediate constant makes the code more efficient without increasing the code size. Extensions to allow 64-bit immediate constants are possible at the cost of having instructions with triple length. However, this feature is not required in the basic ForwardCom design because the priority has been to minimize the number of different instruction sizes for reasons explained below.

2.2 Instruction size

The ForwardCom instruction set uses a 32-bit word size for code. An instruction can consist of one or two 32-bit words, with possible extensions to three or more words. The code density can be increased by using tiny instructions of half the size, but the 32-bit unit size is preserved by pairing tiny instructions two-by-two. It is not possible to jump to the second tiny instruction in such a pair of tiny instructions. It is possible to add future extensions with instruction sizes of three or more words.

Rationale

A CISC architecture with many different instruction sizes is inefficient in superscalar processors where we want to execute several instructions per clock cycle. The decoding front end is often a bottleneck. You have to determine the length of the first instruction before you know where the next instruction begins. The "instruction length decoding" is a fundamentally serial process which makes it difficult to decode multiple instructions per clock cycle. Some microprocessors have an extra "micro-operations cache" after the decoder in order to circumvent this bottleneck.

Here, it is desired to have as few different instruction lengths as possible and to make it easy to determine the length of each instruction. We want a small instruction size for the most common simple instructions, but we also need a larger instruction size in order to accommodate things like a larger register set, instructions with multiple operands, vector operations with advanced features, 32-bit address offsets, and large immediate constants. This proposal is a compromise between code compactness, easy decoding, and space for advanced features.

2.3   Register set

There are 32 general purpose registers (r0–r31) of 64 bits each, and 32 vector registers (v0–v31) of variable length. The maximum vector length is different for different hardware implementations. The general purpose registers can be used for integers of up to 64 bits as well as for pointers. The vector registers can be used for scalars or vectors of integers and floating point numbers.

The following special registers are defined and visible at the application program level. All have 64 bits:

- Instruction pointer (IP)

- Data section pointer (DATAP)

- Thread environment block pointer (THREADP)

- Stack pointer (SP)

- Numeric control register (NUMCONTR)

The stack pointer is identical to r31. The other special registers cannot be accessed as ordinary registers.

There is no dedicated flags register. Registers r1–r7 and v1–v7 can be used for masks, predicates and floating point option flags to control attributes such as rounding mode and exception control.

The unused part of a register is always set to zero. This means that integer operations with an operand size smaller than 64 bits and vector operations with a vector length smaller than the maximum will always set the unused bits of the destination register to zero.

Rationale

The number of registers is a compromise between code density and flexibility. The cost of spilling registers to memory is usually important only in the critical innermost loop, which is unlikely to need more than 32 registers.

We can avoid false dependencies on the previous value of a register by setting all unused register bits to zero rather than leaving them unchanged. The hardware can save power by disabling the unused parts of execution units and data buses.

A dedicated flags register is unfeasible for code that schedules multiple calculations in between each other and for vector code.

The reason for handling floating point scalars in the vector registers rather than in separate registers is to make it easy for a compiler to convert scalar code including function calls to vector code. Floating point code often contains calls to mathematical library functions. If a library function has variable-length vectors as input and output then the same function can be used for both scalars and vectors, and the compiler can easily vectorize code that contains such library function calls.

2.4 Vector support

A vector register can contain integers of 8, 16, 32, 64, and optionally 128 bits, or floating point numbers of single, double, and optionally quadruple precision. All elements of a vector must have the same type. The elements of a vector are processed in parallel. For example, a vector addition will produce the sum of two vectors in a single operation.

The vector registers have variable length. Each vector register has extra bits for storing the length of the vector. The maximum vector length depends on the hardware. For example, if the hardware supports a maximum vector length of 64 bytes and a particular application needs only 16 bytes, then the vector length is set to 16.

Some instructions need to specify the length of a vector explicitly, for example when reading a vector from memory. These instructions use a general purpose register for specifying the vector length. The length is usually indicated as the number of bytes, not the number of vector elements.

A special register gives information about the maximum vector length. The maximum length supported by the processor must be a power of 2. The actual length specified does not have to be a power of 2. If the specified length is longer than the maximum length, then the maximum length is used.

The contents of a vector register can arbitrarily be interpreted as any of the types and element sizes supported. For example, the hardware does not prevent the application of integer instructions on a vector that contains floating point data. It is the responsibility of the programmer that the code makes sense.

2.5 Vector loops

A special addressing mode is provided to make vector loops more compact. It uses a base pointer P and a negative index J and calculates the address of a memory operand as P-J, where P and J are general purpose registers. This makes it possible to make a loop through an array as illustrated by the following pseudocode:

```
P = address of array
J = size of array (in bytes)
L = maximum vector length (depends on processor)
X = a vector register
P += J; // point to end of array
while (J > 0) {
    X = whatever_operation(X),[P−J],(vector length J)
    J −= L;
}
```

This loop works in the following way: P points to the end of the array. J is the remaining number of array elements; counting down until the loop is finished. The loop reads one vector at a time from the array at the address [P-J]. J is larger than the maximum vector length L in all but the last iteration of the loop. This makes the processor use the maximum vector length. If the array size is not divisible by the maximum vector length then the last iteration of the loop will use a smaller vector length that fits the remaining number of elements. Obviously, the loop can contain any number of vector read, vector write, and vector arithmetic instructions, using the same principle.

This loop will work on different processors with different maximum vector lengths *without knowing the maximum vector length at compile time*. Thus, the same piece of software will work on different microprocessors with different vector lengths without the need to compile separately for each microprocessor. A further advantage is that no extra code is needed after the loop to handle remaining elements in the case that the array size is not divisible by the vector length.

Rationale

Most current systems have fixed vector lengths. If different processors have different vector lengths then you have to compile the code separately for each vector length. Every time a new processor with longer vectors comes on the market, you have to compile a new version of the code for the new vector length, using newly defined extensions to the instruction set. It usually takes several years for the new software to be developed and to penetrate the mainstream market. It is so costly for software producers to develop and maintain different versions of their code for each vector length that this is rarely done.

A further problem with current systems is that it is impossible to save a vector register in a way that is guaranteed to be compatible with future processors with longer vectors. This is no problem with the ForwardCom design because the vector length is stored in the vector register. Instructions are provided for saving and restoring vectors of variable length and for storing only the part of a vector register that is actually used.

The ForwardCom design makes it possible to take advantage of a new processor with a longer vector registers immediately without recompiling the code. The loop method described above makes this easy and very efficient. You don't need different versions of the code for different processors.

It is possible to obtain the same effect without the special negative addressing mode by inverting the sign of J and allowing a negative value in the register that specifies the vector length while using the absolute value for the actual vector length. This solution is less elegant and more confusing, but it may possibly be included in the ForwardCom design by allowing negative values when specifying a vector length.

Loop unrolling is generally not necessary. The loop overhead is already reduced to a single instruction (subtract and jump if positive) and a superscalar processor will execute multiple iterations in parallel if dependency chains are not too long. Loop unrolling with multiple accumulators may be useful for hiding a loop-carried dependency. In this case, you will either insert a loop control instruction after each section in the unrolled code or calculate the loop iteration count before the loop.

The ForwardCom design has no practical limit to the vector length that a microprocessor can support. A large microprocessor with very long vectors can be useful for calculations with a high amount of data parallelism. Other solutions to obtain high performance on parallel data processing have been discussed, such as rolling register stacks and software pipelining, but it was concluded that long vectors is the method that can be implemented most efficiently in the microprocessor as well as in the compiler.

2.6 Maximum vector length

The maximum length of vector registers will be different for different processors. The maximum length must be a power of 2. It can be as large as desired and must be at least 16 bytes. Each instruction can use a smaller length, which does not need to be a power of 2.

The maximum length may be different for different element sizes. For example, the maximum length for 32-bit integers can be 32 bytes to contain eight integers, while the maximum length for 8-bit integers could be 16 bytes to contain 16 smaller numbers. However, the maximum length must be the same for different types with the same element size. For example, the maximum length for double precision floating point numbers must be the same as for 64-bit integers because loops are likely to contain both types when integer vectors are used as masks for floating point vectors. The maximum

length for a 32-bit element size cannot be less than for any other element size or operand type. This rule guarantees that it is possible to save a complete vector using a 32-bit operand type.

The maximum vector length should generally be the same for all instructions for the same data type, but there may be exceptions for instructions that are particularly expensive to implement.

A few special registers give information about the maximum vector length supported by the hardware for each vector element size. It is possible for an application program or the operating system to reduce the maximum vector length. This can be useful if a smaller vector length is more appropriate for a particular purpose.

It is also possible to increase the apparent maximum vector length for purposes of emulation. Virtual vector registers that are bigger than what the hardware supports can be emulated through traps (synchronous interrupts) in order to verify the functionality of a program on processors with a longer maximum vector length than is currently available.

When an instruction specifies a longer vector than the maximum, then the maximum length is used unless the emulation of larger vectors is activated. This is necessary for the efficient implementation of vector loops as described above on page 9.

2.7 Instruction masks

Most instructions can have a mask register which can be used for conditional execution and for specifying various options. Instructions with general purpose registers use one of the registers r1–r7 as a mask register or predicate. Bit 0 of the mask register indicates whether the operation is executed or not. Bit 1 of the mask register indicates whether the result should be zero or unchanged in case the operation is not executed.

This mechanism can be vectorized. Instructions with vector registers use one of the vector registers v1–v7 as mask register. The calculation of each vector element is conditional on the corresponding element in the mask register.

Additional bits in the mask register are used for various options, overriding the values in the numeric control register.

2.8 Addressing modes

All memory addressing is relative to some base pointer. Memory operands can be addressed in one of the two general forms:

Address = BP + IX∗SF
Address = BP + OS

Where BP is a 64-bit base pointer, IX is a 64-bit index register, SF is a scale factor, and OS is a direct offset. A base pointer is always present; the other elements are optional.

The base pointer, BP, can be a general purpose register, or it can be the data section pointer (DATAP), instruction pointer (IP) or stack pointer (SP).

The index register, IX, can be one of the registers r0–r30. A value of 31 means no index register.

A limit can be applied to the index register in the form of a 16-bit unsigned integer. A trap is generated if the index register is bigger than the limit in an unsigned comparison.

The scale factor, SF, is equal to the operand size (in bytes) for scalar operands and broadcasts. The scale factor is 1 for vector operands. A special addressing mode with SF = -1 is also available, as explained on page 9.

The offset, OS, is a sign-extended integer of 8, 16 or 32 bits. 8-bit offsets are multiplied by the operand size. Offsets of 16 and 32 bits have no multiplier.

Support for addressing modes with both index and offset is optional.

Jumps and calls specify a target address relative to the instruction pointer. The relative address is specified with a signed offset of 8, 16, 24, or 32 bits, multiplied by the code word size which is 4. This will cover an address range of $\pm$ 8 gigabytes with the 32-bit offset.

Rationale

A 64-bit address space is used. Relative addressing is used in order to avoid 64-bit addresses in the instruction code. In the rare case that a 64-bit absolute address is needed, it must be loaded into a register which is then used as a pointer.

Addressing with an index scaled by the operand size is useful for arrays. A limit can be applied to the index so that array bounds can be checked without any extra instructions.

Addressing with a negative index is useful for the efficient implementation of vector loops described on page 9.

The addressing modes specified here will cover all common applications, including arrays, vectors, structures, classes, and stack frames.

Support for addressing modes with both base pointer, index and direct offset is optional because this requires two adders in the address-calculation stage in the pipeline which might limit the maximum clock frequency.

Chapter 3

Instruction formats

3.1 Formats and templates

All instructions use one of the general format templates shown below (the most significant bits are shown to the left). The basic layout of the 32-bit code word is shown in template A. Template B, C and D are derived from template A by replacing 8, 16 or 24 bits, respectively, with immediate data. Double-size and triple-size instructions can be constructed by adding one or two 32-bit words to one of these templates. For example, template A with an extra 32-bit word containing data is called A2. Template E2 is an extension to template A where the second code word contains an extra register field, extra opcode bits, option bits, and data.

Some small, often-used instructions can be coded in a tiny format that uses a half code word. Two such tiny instructions can be packed into a single code word, using template T. An unpaired tiny instruction must be combined with a tiny-size NOP to fill a whole code word.

Bits	2	3	6	5	1	2	5	3	5
Field	IL	Mode	OP1	RD	M	OT	RS	Mask	RT
Template A. Has three operand registers and a mask register.									

Bits	2	3	6	5	1	2	5	8
Field	IL	Mode	OP1	RD	M	OT	RS	IM1
Template B. Has two operand registers and an 8-bit immediate constant.								

Bits	2	3	6	5	8	8
Field	IL	Mode	OP1	RD	IM2	IM1
Template C. Has one operand register two 8-bit immediate constants.					

Bits	2	3	3	24
Field	IL	Mode	OP1	IM2
Template D. Has no register and a 24-bit immediate constant.			

Bits	2	3	6	5	1	2	5	3	5
Field	IL	Mode	OP1	RD	M	OT	RS	Mask	RT
Field	OP2		OP3	RU	IM2				
Template E2. Has 4 register operands, mask, a 16-bit immediate constant and extra bits for opcode or options.									

Bits	2	3	6	5	1	2	5	3	5
Field	IL	Mode	OP1	RD	M	OT	RS	Mask	RT
Field	IM2								
Template A2. 2 words. As A, with an extra 32-bit immediate constant.									

Bits	2	3	6	5	1	2	5	3	5
Field	IL	Mode	OP1	RD	M	OT	RS	Mask	RT
Field	IM2								
Field	IM3								
Template A3. 3 words. As A, with two extra 32-bit immediate constants.									

Bits	2	3	6	5	1	2	5	8
Field	IL	Mode	OP1	RD	M	OT	RS	IM1
Field	IM2							
Template B2. As B, with an extra 32-bit immediate constant.								

Bits	2	3	6	5	1	2	5	8
Field	IL	Mode	OP1	RD	M	OT	RS	IM1
Field	IM2							
Field	IM3							
Template B3. As B, with two extra 32-bit immediate constants.								

Bits	2	3	6	5	8	8
Field	IL	Mode	OP1	RD	IM2	IM1
Field	IM3					
Template C2. As C, with an extra 32-bit immediate constant.						

Bits	4	14	14
Field	0111	Tiny instruction 2	Tiny instruction 1
Template T. 1 word containing two tiny instructions.			

Bits	5	5	4
Field	OP1	RD	RS
Format for each tiny instruction			

The meaning of each field is described in the following table.

Table 3.13: Fields in instruction templates

Field name	Meaning	Values
IL	Instruction length	0 or 1: 1 word = 32 bits 2: 2 words = 64 bits 3: 3 or more words
Mode	Format	Determines the format template and the use of each field. Extended with the M bit when needed. See details below.

OP1	Opcode	Decides the operation, for example add or move.
OT	Operand type and size (OS)	0: 8 bit integer, OS = 1 byte 1: 16 bit integer, OS = 2 bytes 2: 32 bit integer, OS = 4 bytes 3: 64 bit integer, OS = 8 bytes 4: 128 bit integer, OS = 16 bytes (optional) 5: single precision float, OS = 4 bytes 6: double precision float, OS = 8 bytes 7: quadruple precision float, OS = 16 bytes (optional) The OT field is extended with the M bit when needed.
RD	Destination register	r0 − r31 or v0 − v31. Also used for first source operand if the instruction format does not specify enough operands.
RS	Source register	r0 − r31 or v0 − v31. Source register, pointer, index or vector length register.
RT	Source register	r0 − r31 or v0 − v31. Source register or pointer.
RU	Source register	r0 − r31 or v0 − v31. Source register.
Mask	mask register	0 means no mask. 1-7 means that a general purpose register or vector register is used for mask and option bits.
M	Operand type or mode	Extends the mode field when bit 1 and bit 2 of Mode are both zero (general purpose registers). Extends the OT field otherwise (vector registers).
OP2	Opcode	Opcode extension.
IM1, IM2, IM3	Immediate data	8, 16, 32 or 64 bits immediate operand or address offset or option bits. Adjacent IM fields can be merged.
OP3	Options	Option bits, mode bits, or immediate data.

Instructions have several different formats, defined by the IL and mode bits, according to the following table

Table 3.14: List of instruction formats

Format name	IL	Mode	Template	Use
0.0	0	0	A	Three general purpose register operands (RD, RS, RT).
0.1	0	1	B	Two general purpose registers (RD, RS) and an 8-bit immediate operand (IM1).
0.2	0	2	A	Three vector register operands (RD, RS, RT).
0.3	0	3	B	Two vector registers (RD, RS) and a broadcast 8-bit immediate operand (IM1).
0.4	0	4	A	One vector register (RD), a memory operand with pointer (RT) and vector length specified by a general purpose register (RS).

0.5	0	5	A	One vector register (RD), a memory operand with base pointer (RT). Negative index and vector length are both specified by RS. This is used for vector loops as explained on page 9.
0.6	0	6	A	One vector register (RD) and a scalar memory operand with base pointer (RT) and index (RS) scaled by operand size.
0.7	0	7	B	One vector register (RD) and a scalar memory operand with base pointer (RS) and 8-bit offset.
0.8	0	0 M=1	A	One general purpose register (RD) and a memory operand with base pointer (RT) and index (RS) scaled by operand size.
0.9	0	1 M=1	B	One general purpose register (RD) and a memory operand with base pointer (RT) and 8-bit offset.
1.0	1	0	A	Single-format instructions. Three general purpose register operands.
1.1	1	1	C	Single-format instructions. One general purpose register and a 16-bit immediate operand.
1.2	1	2	A	Single-format instructions. Three vector register operands.
1.3	1	3	B, C	Single-format instructions. Two vector registers and a broadcast8-bit immediate operand, or one vector register and a broadcast 16-bit immediate operand.
1.4	1	4	B	Jump instructions with two register operands and 8 bit offset.
1.5	1	5	C, D	Jump instructions with one register operand, 8 bit constant (IM2) and 8 bit offset (IM1), or no register and 24 bit offset.
1.8	1	0 M=1	B	Single-format instructions. Two general purpose registers and an 8-bit immediate operand.
T	1	6-7	T	Two tiny instructions.
2.0	2	0	A2	Two general purpose registers (RD, RS) and a memory operand with base (RT) and 32 bit offset (IM2).
2.1	2	1	A2	Three general purpose registers and a 32-bit immediate operand IM2.
2.2	2	2	A2	One vector register (RD) and a memory operand with base (RT) and 32 bit offset (IM2). Vector length specified by general purpose register RS.
2.3	2	3	A2	Three vector registers and a broadcast 32-bit immediate operand IM2.
2.4.0	2	4	E2	OP3=00xxxx. Two vector registers (RD, RU) and a scalar memory operand with base (RT) and 16 bit offset (IM2), broadcast to length (RS).
2.4.1	2	4	E2	OP3=01xxxx. Two vector registers (RD, RU) and a memory operand with base (RT), 16 bit offset (IM2), length (RS).

2.4.2	2	4	E2	OP3=10xxxx. Two vector registers (RD, RU) and a memory operand with base (RT), negative index (RS), and length (RS). Optional support for offset IM2 ≠ 0, otherwise IM2 = 0.
2.4.3	2	4	E2	OP3=11xxxx. Two vector registers (RD, RU) and a scalar memory operand with base (RT), scaled index (RS), and limit RS ≤ IM2 (unsigned).
2.5	2	5	E2	Three vector registers (RD, RS, RT) and a broadcast 16-bit immediate integer IM2. IM2 is shifted left by the 6-bit unsigned value of OP3, unless OP3 is used for other purposes. RU is usually unused.
2.6	2	6	A2	Single-format instructions. Three general purpose registers and a 32-bit immediate operand.
2.7	2	7	A2, B2, C2	Jump instructions (OP1 < 16). Single-format instructions. Three vector registers and a 32-bit immediate operand.
2.8.0	2	0 M=1	E2	OP3=00xxxx. Three general purpose registers (RD, RS, RU) and a memory operand with base (RT) and 16 bit offset (IM2).
2.8.1	2	0 M=1	E2	OP3=01xxxx. Two general purpose registers (RD, RU) and a memory operand with base (RT), index (RS), no scale. Optional support for offset IM2 ≠ 0, otherwise IM2 = 0.
2.8.2	2	0 M=1	E2	OP3=10xxxx. Two general purpose registers (RD, RU) and a memory operand with base (RT), scaled index (RS). Optional support for offset IM2 ≠ 0, otherwise IM2 = 0.
2.8.3	2	0 M=1	E2	OP3=11xxxx. Two general purpose registers (RD, RU) and a memory operand with base (RT), scaled index (RS), and limit RS ≤ IM2 (unsigned).
2.9	2	1 M=1	E2	Three general purpose registers (RD, RS, RT) and a 16-bit immediate integer IM2. IM2 is shifted left by the 6-bit unsigned value of OP3, unless OP3 is used for other purposes. RU is usually unused.
3.0	3	0	A3, B3	Jump instructions. Single-format instructions with general purpose register operands. Optional.
3.1	3	1	A3	Three general purpose registers and a 64-bit immediate operand. Optional.
3.2	3	2	A3	Single-format vector instructions. Optional.
3.3	3	3	A3	Three vector registers and a broadcast 64-bit immediate operand. Optional.
3.8	3	0 M=1		Currently unused.
4.x	3	4-7		Reserved for future 4-word instructions and longer.

3.2 Coding of operands

Operand type

The type and size of operands is determined by the OT field as indicated above. The operand type is 64 bit integer (OS = 8) by default if there is no OT field.

Register type

The instructions can use either general purpose registers or vector registers. General purpose registers are used for source and destination operands and for masks if mode is 0 or 1 (with M = 0 or 1). Vector registers are used for source and destination operands and for masks if mode is 2-7. A value of zero in the mask field indicates no mask and unconditional operation.

Pointer register

Instructions with a memory operand always use an address relative to a base pointer. The base pointer can be a general purpose register, the data section pointer, or the instruction pointer. The pointer is determined by the RS or RT field. This field is interpreted as follows.

Instruction formats with no offset or 8-bit offset (0.4-0.9) can use any of the registers r0-r31 as base pointer. r31 is the stack pointer.

Instruction formats with 16-bit or 32-bit offset (2.0, 2.2, 2.4, 2.8) can use the same registers, except r29 which is replaced by the data section pointer (DATAP), and r30 which is replaced by the instruction pointer (IP). This also applies to formats with an unused 16-bit offset (format 2.4.2 and 2.4.3).

Tiny instructions with a memory operand can use r0-r14 or the stack pointer (r31) as pointer in the 4-bit RS field. A value of 15 in the RS field indicates the stack pointer.

Index register

Instruction formats with an index can use r0-r30 as index. A value of 31 in the index field (RS) means no index. The signed index is multiplied by the operand size (OS) for formats 0.6, 0.8, 2.4.3, 2.8.2, 2.8.3; by 1 for format 2.8.1; or by -1 for format 0.5 and 2.4.2. The result is added to the value of the base pointer.

Offsets

Offsets can be 8, 16 or 32 bits. The value is sign-extended to 64 bits. An 8-bit offset is multiplied by the operand size OS, as given by the OT field. An offset of 16 or 32 bits is not scaled. The result is added to the value of the base pointer.

Support for addressing modes with both index and offset is optional (format 2.4.2, 2.8.1, 2.8.3). If this kind of addressing involving two additions is not supported then the offset in IM2 must be zero.

Limit on index

Formats 2.4.3 and 2.8.3 have a 16-bit limit on the index register. This is useful for checking array limits. If the value of the index register, interpreted as unsigned, is bigger than the unsigned limit then a trap is generated.

Vector length

The vector length of memory operands is specified by r0-r30 in the RS field for formats 0.4, 0.5, 2.2, 2.4. A value of 31 in the RS field indicates a scalar with the same length as the operand size (OS).

The value of the vector length register gives the vector length of the memory operand in bytes (not the number of elements). If the value is bigger than the maximum vector length then the maximum vector length is used. The value may be zero. The behavior for negative values is implementation dependent: either interpret the value as unsigned or use the absolute value.

The vector length must be a multiple of the operand size OS, as indicated by the OT field. If the vector length is not a multiple of the operand size then the behavior of the partial vector element is implementation dependent.

The vector length for source operands in vector registers is saved in the register.

Combining vectors with different lengths

The vector length of the destination will be the same as the vector length of the first source operand (even if the first source operand uses the RD field).

A consequence of this is that the length of the result is determined by the order of the operands when two vectors of different lengths are combined.

If the source operands have different lengths then the lengths will be adjusted as follows. If a vector source operand is too long then the extra elements will be ignored. If a vector source operand is too short then the missing elements will be zero.

A scalar memory operand (format 0.6 and 0.7) is not broadcast but treated as a short vector. It is padded with zeroes to the vector length of the destination.

A broadcast memory operand (format 2.4.1) will use the vector length given by the vector length register in the RS field.

A broadcast immediate operand will use the same vector length as the destination.

Immediate constants

Immediate constants can be 4, 8, 16, 32, and optionally 64 bits. Immediate fields are generally aligned to natural addresses. They are interpreted as follows.

If OT specifies an integer type then the field is interpreted as an integer. If the field is smaller than the operand size then it is sign-extended to the appropriate size. If the field is larger than the operand size then the superfluous upper bits are ignored. The truncation of a too large immediate operand will not trigger any overflow condition.

If OT specifies a floating point type then the field is interpreted as follows. Immediate fields smaller than 32 bits are interpreted as signed integers and converted to floating point numbers of the desired precision. A 32-bit field is interpreted as a single precision floating point number. It is converted to the desired precision if necessary. A 64-bit field (if supported) is interpreted as a double precision floating point number. A 64-bit field is not allowed with a single precision operand type. A few optional instructions in format 1.3C have a half-precision floating point immediate constant that is converted to a single or double precision scalar.

16-bit immediate constants in format 2.5 and 2.9 can be shifted left by the 6-bit unsigned value of OP3 to give a 64-bit signed value. Any overflow beyond 64 bits is ignored. The shift is done before any conversion to floating point. No shifting is done if OP3 is used for other purposes.

An instruction can be made compact by using the smallest immediate field size that fits the actual value of the constant.

Mask register

The 3-bit mask field indicates a mask register. Register r1-r7 is used if the destination is a general purpose register. Vector register v1-v7 is used if the destination is a vector register. A value of zero in

the mask field means no mask and unconditional execution using the options specified in the numeric control register.

If the mask is a vector register then it is interpreted as a vector with the same element size as indicated by the OT field. Each element of the mask register is applied to the corresponding element of the result.

The meanings of the flag bits are described in the next section.

3.3   Coding of masks

A mask register can be a general purpose register r1-r7 or a vector register v1-v7. A value of zero in the mask field means no mask.

The bits in the mask register are coded as follows.

Table 3.15: Bits in mask register and numeric control register

Bit number	Meaning
0	Predicate or mask. The operation is executed only if this bit is one. If this bit is zero then the operation is not executed, and any arithmetic error condition is suppressed.
1	Zeroing. This bit determines the result when bit 0 is zero. Bit 1 = 0 makes the result zero. Bit 1 = 1 makes the value unchanged, i. e. the output is the same as the input from the first source operand. Bit 1 has no effect when bit 0 = 1.
2	Detect unsigned integer overflow.
3	Detect signed integer overflow.
6	Propagate error bits detected by bit 2 or 3. This feature is tentative, see page 63.
7	Generate a trap if overflow as indicated by bit 2 or 3 is detected.
18-19	Floating point rounding mode: 00 = nearest or even 01 = down 10 = up 11 = towards zero
20	Support subnormal numbers. Subnormal floating point numbers are treated as zero or flushed to zero when this bit is 0 (this is generally faster).
22	Better NAN propagation. If this bit is zero then the IEEE Standard 754-2008 (or later) is followed strictly for NAN values. A value of one in bit 22 improves NAN propagation and the use of NANs for tracing floating point errors. The details are described on page 62.
26	Enable trap on floating point overflow and division by zero.
27	Enable trap on floating point invalid operation.
28	Enable trap on floating point underflow and precision loss.

29	Enable trap for NAN inputs to compare instructions and floating point to integer conversion instructions.

Bits 8-9, 16-17, 24-25, etc. in a vector mask register can be used like bits 0-1 for 8-bit and 16-bit operand sizes. All other bits are reserved for future use.

Vector instructions treat the mask register as a vector with the same element size (OS) as the operands. Each element of the mask vector has the bit codes as listed above. The different vector elements can have different mask bits.

The numeric control register (NUMCONTR) is used as mask when the mask field is zero or absent. The NUMCONTR register is broadcast to all elements of a vector, using as many bits of NUMCONTR as indicated by the operand size, when an instruction has no mask register. The same mask is applied to all vector elements in this case. Bit 0 in NUMCONTR must be 1.

3.4   Format for jump, call and branch instructions

Most branches in ordinary code are based on the result of an arithmetic or logic instruction (ALU). The ForwardCom design combines the ALU instruction and the conditional jump into a single instruction. For example, a loop control can be implemented with a single instruction that counts down and jumps until it reaches zero or counts up through negative values and jumps until it reaches zero.

The jumps, calls, branches and multiway branches will use the following formats.

Table 3.16: List of formats for control transfer instructions

Format	IL	Mode	OP1	Template	Description
1.4	1	4	OPJ	B	Short version with two register operands (RD, RS) and 8 bit offset (IM1).
1.5 C	1	5	OPJ	C	Short version with one register operand (RD), an 8-bit immediate constant (IM2) and 8 bit offset (IM1), or a 16-bit offset (IM2+IM1 combined).
1.5 D	1	5	0-7	D	Jump or call with 24-bit offset.
2.7.0	2	7	0	B2	Double size version with two register operands and 32 bit offset (IM2). IM1 = OPJ.
2.7.1	2	7	1	B2	Double size version with a register destination operand, a register source operand, a 16-bit offset (IM2 lower half) and a 16-bit immediate operand (IM2 upper half).
2.7.2	2	7	2	C2	Double size version with one register operand (RD), one 8-bit immediate constant (IM2) and 32 bit offset (IM3).
2.7.3	2	7	3	C2	Double size version with one register operand (RD), an 8-bit offset (IM2) and a 32-bit immediate constant (IM3).
2.7.4	2	7	4	C2	Double size system call, no OPJ, 16 bit constant (IM1,IM2) and 32-bit constant (IM3).
3.0.0	3	0	0	C2	No operation (NOP).

3.0.1	3	0	1	B3	Triple size version with a register destination operand, a register source operand, a 32-bit immediate operand (IM2) and a 32-bit offset (IM3). Optional.

The jump, call and branch instructions have signed offsets of 8, 16, 24 or 32 bits relative to the instruction pointer. Or, more precisely, relative to the end of the instruction. This offset is multiplied by the instruction word size (= 4) to cover an address range of ± a half kilobyte for short conditional jumps with 8 bits offset, ± 128 kbytes for jumps and calls with 16 bits offset, ± 32 megabytes for 24 bits offset, and ± 8 gigabytes for 32 bits offsets. The optional triple-size format includes unconditional jump and call with a 64 bits absolute address.

The versions with template C and C2 have no OT field. The operand type is 64-bit integer when there is no OT field. It is not possible to use formats with template C or C2 with floating point types. The instructions will use vector registers (first element only) when there is an OT field and M = 1. In other words, the ALU-jump instructions will use vector registers only when a floating point type is specified (or 128-bit integer type, if supported). General purpose registers are used in all other cases. It is possible to use bitwise logical instructions with vector registers by specifying a floating point type.

The OPJ field defines the operation and jump condition. This field has 6 bits in the single size version and 8 bits in the longer versions. The two extra bits in the longer versions are used as follows. Bit 6 is reserved for future extensions, and must be zero. Bit 7 may be used for indicating loop behavior as a hint for choosing the optimal branch prediction algorithm.

The lower 6 bits of the OPJ field contains the following codes.

Table 3.17: List of control transfer instructions: jump, call, return

OPJ	bit 0 of OPJ	Function	Comment
0-7	part of offset	Unconditional jump with 24-bit offset	Format 1.5 D
8-15	part of offset	Unconditional call with 24-bit offset	Format 1.5 D
0-1	invert	Subtract signed, jump if negative (sub_sign_jmpneg)	Format 1.4 and 2.7.0. No floating point.
2-3	invert	Subtract signed, jump if positive (sub_sign_jmppos)	Format 1.4 and 2.7.0. No floating point.
4-5	invert	Subtract unsigned, jump if borrow (sub_unsign_jmpborrow)	Format 1.4 and 2.7.0. No floating point.
6-7	invert	Subtract unsigned, jump if not zero or borrow (sub_unsign_jmpnzc)	Format 1.4 and 2.7.0. No floating point.
8-9	invert	Subtract, jump if not zero (sub_jmpnzero)	Format 1.4 and 2.7.0. No floating point.
10-11	invert	Subtract signed, jump if overflow (sub_sign_jmpovfl)	Format 1.4 and 2.7.0. No floating point.

12-15		Reserved for future use	Format 1.4 and 2.7.0.
16-17	invert	Add signed, jump if negative (add_sign_jmpneg)	No floating point.
18-19	invert	Add signed, jump if positive (add_sign_jmppos)	No floating point.
20-21	invert	Add unsigned, jump if carry (add_unsign_jmpcarry)	No floating point.
20-21	invert	Jump if either operand is ± infinite or NAN (cmp_float_jmpinfnan)	Floating point operands
22-23	invert	Add unsigned, jump if not zero or carry (add_unsign_jmpnzc)	No floating point.
22-23	invert	Jump if either operand is subnormal (cmp_float_jmpsubnorm)	Floating point operands
24-25	invert	Add, jump if not zero (add_jmpnzero)	No floating point
26-27	invert	Add signed, jump if overflow (add_sign_jmpovfl)	No floating point
28-29	invert	Shift left by n, jump if not zero (shift_jmpnzero)	Shift right unsigned if n negative
30-31	invert	Shift left by n, jump if carry (shift_jmpcarry)	Shift right unsigned if n negative
32-33	invert	Compare signed, jump if below (cmp_sign_jmpbelow)	
34-35	invert	Compare signed, jump if above (cmp_sign_jmpabove)	
36-37	invert	Compare unsigned, jump if below (cmp_unsign_jmpbelow)	Integer operands
36-37	invert	Jump if either operand is NAN (cmp_float_jmpunordered)	Floating point operands
38-39	invert	Compare unsigned, jump if above (cmp_unsign_jmpabove)	Integer operands
38-39	invert	Jump if either operand is ± infinite (cmp_float_jmpinf)	Floating point operands
40-41	invert	Compare, jump if not equal (cmp_jmpneq)	
42-43	invert	Bitwise test, jump if not zero (test_jmpnzero)	
44-45	invert	Bitwise and, jump if not zero (and_jmpnzero)	
46-47	invert	Bitwise or, jump if not zero (or_jmpnzero)	
48-49	invert	Bitwise xor, jump if not zero (xor_jmpnzero)	
50-51	invert	Test single bit, jump if not zero (testbit_jmpnzero)	
52-53	invert	Test single bit on vector register, jump if not zero (testbit_jmpnzero)	

54-57		Reserved for future use.	
58-59	0 jump 1 call	Indirect with pointer address in register RS, pointer offset in IM1 or IM2 (jump/call).	Format 1.4 and 2.7.0.
58-59	0 jump 1 call	Unconditional direct jump/call with 16 bit or 32 bit offset or 64-bit absolute address (jump/call)	Format 1.5 C, 2.7.2, and 3.0.1.
60-61	0 jump 1 call	Use table of addresses relative to RD. RT = table base, RS = index*OS (jump/call)	Format 1.4, template A.
60-61	0 jump 1 call	Unconditional jump or call to value of register RS (jump/-call)	Format 1.5.
62	0	Return from function (return)	Format 1.4.
62	0	Return from system function (sys_return)	Format 1.5.
63	1	System call. ID in register RT, shared memory block RD, length RS. No mask (sys_call)	Format 1.4, template A.
63	1	System call. ID in constants, shared memory block RD, length RS (sys_call)	Format 2.7.1, 2.7.4 and 3.0.1.
63	1	Unconditional trap. Interrupt number in IM1 (trap).	Format 1.5.
63	1	Filler for unused code memory. All fields are all 1's (filler).	Format 1.5.
63	1	Trap if unsigned RD > IM3. IM2 = 38. Interrupt number is fixed (cmp_unsign_trapabove).	Format 2.7.3.

Signed integer comparisons are corrected for overflow, but signed addition and subtraction are not. For example, if A is a large positive integer and B is a large negative integer then sub_sign_jmpneg will jump if the calculation of A-B overflows to give a negative result, but cmp_sign_jmpbelow will not jump because A is bigger than B.

The combined ALU and conditional jump instructions can be coded in the formats 1.4, 1.5 C, 2.7.0, 2.7.1, 2.7.2, 2.7.3, and 3.0.1, except subtraction which cannot be coded in format 1.5 C. Subtraction with an immediate constant can be replaced by addition with the negative constant. The code space that would have been used by subtraction in format 1.5 C is instead used for coding direct jump and call instructions with a 24-bit offset using format 1.5 D, where the lower three bits of OP1 are used as part of the 24-bit offset.

The add and subtract operations are usually not supported for floating point operands because the longer latencies of these floating point operations will complicate the pipeline design. Floating point compare is supported because it is possible to make a floating point compare operation in a single clock cycle, using unsigned integer compare on the combined exponent and significant with special handling of the sign bit and NAN values.

The test bit instruction (testbit_jmpnzero) will test bit number n in the first operand, where n is the value of the second operand (RS or IM2). This is useful for testing bit fields, sign bits, and the output of compare instructions. The second operand is interpreted as an 8-bit unsigned integer regardless of the operand type.

The shift left instructions will shift the first operand left when the second operand is positive and shift

right with zero extension when the second operand is negative. The carry is the last bit shifted out. The operands are interpreted as integers regardless of the operand type, but vector registers are used if a floating point operand type is specified (M = 1).

Unconditional and indirect jumps and calls use the formats indicated above, where unused fields must be zero. Bit 0 of the OPJ field is zero for jump instructions and one for call instructions.

The table-based indirect jump/call instructions are intended to facilitate multiway branches (switch/-case statements), function tables in code interpreters, and virtual function tables in object oriented languages with polymorphism. The table of jump or call addresses is stored as signed offsets relative to an arbitrary reference point, which may be the table address, the code base, or any other reference point. The operand type specifies the size of the table entries. 16-bit and 32-bit table entries must be supported. Other sizes are optional. The use of relative addresses makes the table more compact than if 64-bit absolute addresses were used. This instruction works as follows. Calculate the address of a table entry as the base pointer (RT) plus the index (RS) multiplied by the operand size. Read a signed value from this address, and scale by 4. Sign-extend this value to 64 bits, and add the reference point (RD). Jump or call to the calculated address. The array index (RS) is scaled by the operand size, while the table entries are scaled by the instruction word size (4). Support for a mask is optional.

The table used by the table-based jump/call instructions may be placed in the constant data section (CONST). This makes it possible to use the table base as reference point and it improves security by giving read-only access to the table.

Return instructions do not need a stack offset when the calling conventions specified on page 82 are used.

System calls use ID numbers rather than addresses to identify system functions. The ID is the combination of a module ID identifying a particular system module or device driver and a function ID identifying a particular function within this module. The module ID and the function ID are both 16 or 32 bits, so that the combined system call ID is up to 64 bits. The sys_call instruction has the following variants:

Table 3.18: Variants of system call instruction

Format	Operand type	Function ID	Module ID
1.4	32 bit	RT bit 0-15	RT bit 16-31
1.4	64 bit	RT bit 0-31	RT bit 32-63
2.7.1	32 bit	IM2 bit 0-15	IM2 bit 16-31
2.7.4	64 bit	IM21 bit 0-15	IM3 bit 0-31
3.0.1	64 bit	IM2 bit 0-31	IM3 bit 0-31

The sys_call instruction can indicate a block of memory to be shared with the system function. The address of the memory block is pointed to by the register specified in RD and the length is in register RS. This memory block, which the caller must have access rights to, is shared with the system function. The system function will get the same access rights to this block as the calling thread has, i. e. read access and/or write access. This is useful for fast transfer of data between the caller and the system function. No other memory is accessible to both the caller and the called function. If the RD and RS fields are both zero (i. e. indicating register r0) then no memory block is shared. The sys_call instruction in format 2.7.4 cannot have a shared memory block.

Parameters for system functions are transferred in registers, following the same calling conventions as normal functions. The registers used for function parameters are usually different from the registers in the RD, RS and RT fields. Function parameters that do not fit into registers must reside in the shared memory block.

Traps work like interrupts. The unconditional trap has an 8-bit interrupt number in IM1. This is an index into the interrupt vector table, which initially starts at absolute address zero. The unconditional

trap instruction may use IM2 for additional information. The conditional trap is intended for checking array bounds. The interrupt number is fixed (the value has not been decided yet). The conditional trap may optionally support other condition codes in IM2, using the same codes as OPJ in table 3.17.

A trap instruction with all 1's in all fields (opcode 0x6FFFFFFF) can be used as filler in unused parts of code memory.

3.5 Assignment of opcodes

The opcodes and formats for new instructions can be assigned according to the following rules.

- Multi-format instructions. Often-used instructions that need to support many different operand types, addressing modes and formats use most or all of the following formats: 0.0-0.9, 2.0-2.5, 2.8-2.9, and optionally 3.1 and 3.3 if triple-size instructions are supported. The same value of OP1 is used in all these formats. OP2 must be 0. Instructions with few source operands come first.

- Tiny instructions. Only some of the most common instructions are available in tiny versions, as there is only space for 32 tiny instructions. The instructions are ordered according to the number and type of operands, as shown in table 4.6 page 34.

- Control transfer instructions, i. e. jumps, branches, calls and returns, can be coded as short instructions with IL = 1, mode = 4-5, and OP1 = 0-63 or as double-size instructions with IL = 2, mode = 7, OP1 = 0-15, and optionally as triple-size instructions with IL = 3, mode = 0, OP1 = 0-15. See page 21.

- Short single-format instructions with general purpose registers. Use mode 1.0, 1.1, and 1.8, with any value of OP1.

- Short single-format instructions with vector registers. Use mode 1.2 and 1.3 with any value of OP1.

- Double-size single-format instructions with general purpose registers can use mode 2.8 and 2.9 with any value of OP1 and OP2 ≥ 8 (give similar instructions the same value of OP1), and mode 2.6 with any value of OP1.

- Double-size single-format instructions with vector registers can use mode 2.4 and 2.5 with any value of OP1 and OP2 ≥ 8 (give similar instructions the same value of OP1), and mode 2.7 with OP1 in the range 16-63.

- Triple-size single-format instructions with general purpose registers can use mode 3.0 with OP1 in the range 16-63.

- Triple-size single-format instructions with vector registers can use mode 3.2 with any value in OP1.

- Future instructions longer than three 32-bit words are coded with IL = 3, mode = 4-7.

- New options or other modifications to existing instructions can use OP3 bits or mask register bits.

- New addressing modes may be implemented as single-format read and write instructions. New addressing modes or other modifications that apply to all multi-format instructions can use OP3 for option bits. If the bits of OP3 are exhausted then it is possible, as a last resort, to use OP2 values in the range 1-7.

All unused fields must be zero. The instructions with the fewest input operands should preferably have the lowest OP1 codes.

The operands are assigned as follows. The destination operand is a register specified in the RD field. Source operands use register fields RS, RT and RU, unless these fields are used for other purposes (i. e. base pointer, index, vector length). If there is a memory operand or an immediate operand then it will be the last source operand. If the chosen format has fewer source operands than needed for the instruction then RD is used as both destination and the first source operand. If there are still not enough operands then the format cannot be used for the specific instruction. If the format has more operands than needed then any memory operand or immediate operand will be the last source operand, taking precedence over any register operand. Unused operand fields must be zero.

Chapter 4

Instruction lists

The ForwardCom instructions are listed in a comma-separated file instruction_list.csv. This file is intended for use by assemblers, disassemblers, debuggers and emulators. The list is preliminary and subject to possible changes. Please remember to keep the lists in this document and the list in the instruction_list.cvs file synchronized.

The instruction list file has the following fields:

Table 4.1: Fields in instruction list file

Field	Meaning
Name	Name of instruction as used by assembler.
Category	1: single format instruction, 2: tiny instruction, 3: multi-format instruction, 4: jump instruction.
Formats	See table 4.2 below.
Template	Hexadecimal number: 0xA - 0xE for template A - E, 0x1 for tiny template, 0x0 for multiple templates.
Source operands	Number of source operands, including register, memory and immediate operands, but not including mask, option bits, vector length, and index.
OP1	Operation code OP1.
OP2	Additional operation code OP2. Zero if none.
OP3 bits used	Number of bits of OP3 field used for options. OP3 is used for shift count in format 2.5 and 2.9 only if the value specified here is zero.
Operand types general purpose registers	Hexadecimal number indicating required and optional support for each operand type with general purpose registers. See table 4.3 below for meaning of each bit.
Operand types scalar	Hexadecimal number indicating required and optional support for each operand type for scalar operations in vector registers. See table 4.3 below for meaning of each bit.
Operand types vector	Hexadecimal number indicating required and optional support for each operand type for vector operations. See table 4.3 below for meaning of each bit.

Immediate operand type	Type of immediate operand for single-format instructions. See table 4.4 below.
Description	Description of the instruction and comments.

Table 4.2: Meaning of formats field in instruction list file

Category	Interpretation of formats field
1. Single format instruction	Number with three hexadecimal digits. The leftmost digit is the value of the IL field (0-3). The middle digit is he value of mode field or the combined M+mode field (0-9). The rightmost digit is the sub-mode defined by OP3 in mode 2.4.x and 2.8.x or OP1 in mode 2.7.x. Zero otherwise. For example 0x283 means format 2.8.3.
2. Tiny instruction	0 no operands. 1 RD = general purpose destination register, RS = immediate operand. 2 RD = g. p. destination register, RS = g. p. source register. 4 RD = g. p. destination register, RS = pointer to memory source operand. 5 RD = g. p. source register, RS = pointer to memory destination operand. 8 RD = vector destination register, RS unused. 9 RD = vector destination register, RS immediate operand. 10 RD = vector destination register, RS vector source register. 11 RD = vector source register, RS g. p. destination register r0-r14,r31. 12 RD = vector destination register, RS = pointer to memory source operand. 13 RD = vector source register, RS = pointer to memory destination operand.
3. Multi-format instruction	Hexadecimal number composed of one bit for each format supported: 0x0000001 Format 0.0: three general purpose registers. 0x0000002 Format 0.1: two general purpose registers, 8-bit immediate. 0x0000004 Format 0.2: Three vector registers. 0x0000008 Format 0.3: Two vectors, 8-bit immediate. 0x0000010 Format 0.4: One vector, memory operand. 0x0000020 Format 0.5: One vector, memory operand with negative index. 0x0000040 Format 0.6: One vector, scalar memory operand with index. 0x0000080 Format 0.7: One vector, scalar memory operand with 8-bit offset. 0x0000100 Format 0.8: One g. p. register, memory operand with index. 0x0000200 Format 0.9: One g. p. register, memory operand with 8-bit offset. 0x0000400 Format 2.0: Two g. p. registers, memory op. with 32-bit offset. 0x0000800 Format 2.1: Three g. p. registers, 32-bit immediate. 0x0001000 Format 2.2: One vector register, memory op. with 32-bit offset.

	0x0002000	Format 2.3: Three vector registers, 32-bit immediate.
	0x0004000	Format 2.4.0: Two vector reg., scalar memory op. w. 16-bit offset.
	0x0008000	Format 2.4.1: Two vector reg., memory op. with 16-bit offset.
	0x0010000	Format 2.4.2: Two vector reg., memory op. with negative index.
	0x0020000	Format 2.4.3: Two vector reg., scalar mem. op. index and limit.
	0x0040000	Format 2.5: Three vector reg., shifted 16-bit immediate.
	0x0080000	Format 2.8.0: Three g. p. reg., memory op. with 16-bit offset.
	0x0100000	Format 2.8.1: Two g. p. reg., memory op. with unscaled index.
	0x0200000	Format 2.8.2: Two g. p. reg., memory op. with scaled index.
	0x0400000	Format 2.8.3: Two g. p. reg., memory op. with index and limit.
	0x0800000	Format 2.9: Three g. p. reg., shifted 16-bit immediate.
	0x1000000	Format 3.1: Three g. p. registers, 64-bit imm. (optional).
	0x2000000	Format 3.3: Three vector registers, 64-bit imm. (optional).
4. Jump instruction	Hexadecimal number composed of one bit for each format supported:	
	0x001	Format 1.4: Two registers, 8-bit offset.
	0x002	Format 1.5 C: One register, 8-bit immediate, 8-bit offset.
	0x004	Format 1.5 C: 16-bit offset.
	0x008	Format 1.5 D: No register, 24-bit offset.
	0x010	Format 2.7.0: Two registers, 32-bit offset.
	0x020	Format 2.7.1: Two registers, 16-bit immediate, 16-bit offset.
	0x040	Format 2.7.2: One register, 8-bit immediate, 32-bit offset.
	0x080	Format 2.7.3: One register, 32-bit immediate, 8-bit offset.
	0x100	Format 2.7.4: System call, 16-bit function, 32-bit module.
	0x200	Format 3.0.1: Two registers, 32-bit immediate, 32-bit offset.
	0x400	Format 3.0.1: 64-bit absolute address.

Table 4.3: Indication of operand types supported for general purpose registers, scalars in vector registers, or vectors. The value is a hexadecimal number composed of one bit for each operand type supported

0x0001	8-bit integer supported.
0x0002	16-bit integer supported.
0x0004	32-bit integer supported.

0x0008	64-bit integer supported.
0x0010	128-bit integer supported.
0x0020	single precision floating point supported.
0x0040	double precision floating point supported.
0x0080	quadruple precision floating point supported.
0x0100	8-bit integer optionally supported.
0x0200	16-bit integer optionally supported.
0x0400	32-bit integer optionally supported.
0x0800	64-bit integer optionally supported.
0x1000	128-bit integer optionally supported.
0x2000	single precision floating point optionally supported.
0x4000	double precision floating point optionally supported.
0x8000	quadruple precision floating point optionally supported.

Table 4.4:   Immediate operand type for single-format instructions

0	none or multi-format.
1	4-bit signed integer.
2	8-bit signed integer.
3	16-bit signed integer.
4	32-bit signed integer.
5	64-bit signed integer.
6	8-bit signed integer shifted by specified count.
7	16-bit signed integer shifted by specified count.
8	16-bit signed integer shifted by 16.
9	32-bit signed integer shifted by 32.
17	4-bit unsigned integer.
18	8-bit unsigned integer.
19	16-bit unsigned integer.
20	32-bit unsigned integer.
21	64-bit unsigned integer.
33	4-bit signed integer converted to float.
34	8-bit signed integer converted to float.
35	16-bit signed integer converted to float.
39	16-bit signed integer shifted by specified count, converted to float.
64	half precision floating point.
65	single precision floating point.
66	double precision floating point.

Jump instructions are listed on page 22. All other categories of instructions are listed in the following tables.

4.1   List of multi-format instructions

The following list covers general instructions that can be coded in most or all of the formats assigned to multi-format instructions.

Table 4.5:   List of multi-format instructions

Instruction	OP1	Source operands	Description
nop	0	0	No operation.
move	1	1	Copy value.
store	2	1	Store value to memory.
prefetch	3	1	Prefetch from memory.
sign_extend	4	1	Sign-extend smaller integer to 64 bits.
add	8	2	src1 + src2.
sub	9	2	src1 - src2.
sub_r	10	2	src2 - src1.
compare	11	2	Compare. Uses condition codes, see p. 45.
mul	12	2	src1 · src2.
mul_hi_s	13	2	(src1 · src2) >> OS, signed (integer only).
mul_hi_u	14	2	(src1 · src2) >> OS, unsigned (integer only).
mul_ex_s	15	2	Multiply even-numbered signed integer vector elements to double size result.
mul_ex_u	16	2	Multiply even-numbered unsigned integer vector elements to double size result.
div	17	2	src1 / src2 (optional for integer vectors).
rem	18	2	Modulo (optional for integer vectors).
min	20	2	Signed minimum.
max	21	2	Signed maximum.
min_u	22	2	Minimum. unsigned for integers, abs for f.p.
max_u	23	2	Maximum. unsigned for integers, abs for f.p.
and	32	2	src1 & src2.
and_not	33	2	src1 & (~src2).
or	34	2	src1 | src2.
xor	35	2	src1 ^ src2.
shift_left	36	2	src1 << src2.
shift_rightu	37	2	src1 >> src2, zero extended.
shift_rights	38	2	src1 >> src2, sign extended.
rotate	39	2	Rotate left if src2 positive, right if negative.
extract_bit	40	2	Extract bit. (src1 >> src2) & 1.
set_bit	41	2	Set bit. src1 | (1 << src2).
clear_bit	42	2	Clear bit. src1 & ~ (1 << src2).
toggle_bit	43	2	Toggle bit. src1 ^ (1 << src2).
mul_add	46	3	± src1 ± src2 · src3 (optional).
add_add	47	3	± src1 ± src2 ± src3 (optional).
userdef55 - userdef62	55-62	2	Reserved for user-defined instructions.
undef	63	2	Undefined code. Guaranteed to generate trap in all future implementations.

4.2   List of tiny instructions

Tiny instructions are fitted two in one 32-bit code word. If a tiny instruction cannot be paired with anything else, it must be paired with a tiny nop.

Tiny instructions have an operand size of 64 bits unless otherwise noted. RD is the destination register, and in most cases also the first source register. RS can be a register r0-r15, v0-v15, or an immediate sign-extended 4-bit constant. Instructions with a pointer in RS use register r0-r14 as pointer when RS is 0-14, and the stack pointer (r31) when RS is 15.

It is not possible to jump to the second instruction in a tiny pair because instruction addresses must be divisible by four. If an interrupt or trap occurs in a tiny instruction then the interrupt handler must remember which of the two tiny instructions in the pair was interrupted.

Table 4.6: List of tiny instructions with general purpose registers

Instruction	OP1	Description
nop	0	No operation.
move	1	RD = sign-extended constant RS.
add	2	RD += sign-extended constant RS.
sub	3	RD −= sign-extended constant RS.
shift_left	4	RD <<= unsigned constant RS.
shift_rightu	5	RD >>= unsigned constant RS (zero extended).
move	8	RD = register operand RS.
add	9	RD += register operand RS.
sub	10	RD −= register operand RS.
and	11	RD &= register operand RS.
or	12	RD |= register operand RS.
xor	13	RD ^= register operand RS.
move	14	Read RD from memory operand with pointer RS (RS = r0-r14, r31).
store	15	Write RD to memory operand with pointer RS (RS = r0-r14, r31).

Table 4.7: List of tiny instructions with vector registers

Instruction	OP1	Description
clear	16	Clear register RD by setting the length to zero.
move	17	RD = signed 4-bit integer RS, converted to single precision scalar.
move	18	RD = signed 4-bit integer RS, converted to double precision scalar.
move	19	RD = RS. Copy vector of any type.
add	20	RD += RS, single precision float vector.
add	21	RD += RS, double precision float vector.
sub	22	RD −= RS, single precision float vector.
sub	23	RD −= RS, double precision float vector.
mul	24	RD *= RS, single precision float vector.
mul	25	RD *= RS, double precision float vector.
add_cps	28	Get size of compressed image for RD and add it to general purpose register RS.
sub_cps	29	Get size of compressed image for RD and subtract it from general purpose register RS.
restore_cp	30	Restore vector register RD from compressed image pointed to by RS.
save_cp	31	Save vector register RD to compressed image pointed to by RS.

4.3 List of single-format instructions

These instructions are mostly available in only one or a few formats.

Table 4.8: List of single-format instructions with general purpose registers

Instruction	Format	OP1	Description
bitscan_f	1.0	1	Bit scan forward. Find index to lowest set bit in RS (optional).
bitscan_r	1.0	2	Bit scan reverse. Find index to highest set bit in RS (optional).
round_d2	1.0	3	Round down RS to nearest power of 2.
round_u2	1.0	4	Round up RS to nearest power of 2.
move	1.1	0	Move 16-bit sign-extended constant to general purpose register RD.
move_u	1.1	1	Move 16-bit zero-extended constant to general purpose register RD (can be used as first step of loading a 32-bit constant if double size instructions are not supported).
add	1.1	2	Add 16-bit sign-extended constant to RD.
sub	1.1	3	Subtract 16-bit sign-extended constant from RD.
subr	1.1	4	Subtract RD from 16-bit sign-extended constant.
mul	1.1	5	Multiply RD with 16-bit sign-extended constant.
div	1.1	6	Divide RD with 16-bit sign-extended constant.
add	1.1	7	Shift 16-bit signed constant left by 16 and add to RD.
move	1.1	16	RD = IM2 << IM1. Sign-extend IM2 to 64 bits and shift left by the unsigned value IM1.
add	1.1	17	RD += IM2 << IM1. Sign-extend IM2 to 64 bits, shift left by the unsigned value IM1, add to RD.
and	1.1	18	RD &= IM2 << IM1. Sign-extend IM2 to 64 bits, shift left by the unsigned value IM1, AND with RD.
or	1.1	19	RD |= IM2 << IM1. Sign-extend IM2 to 64 bits, shift left by the unsigned value IM1, OR with RD.
xor	1.1	20	RD ^= IM2 << IM1. Sign-extend IM2 to 64 bits, shift left by the unsigned value IM1, XOR with RD.
abs	1.8	0	Absolute value of integer. Use saturation if IM1 = 1.
shift_add	1.8	1	Shift and add. RD += RS << IM1 (shift right zero extended if IM1 negative).
read_spe	1.8	32	Read special register RS into g. p. register RD.
write_spe	1.8	33	Write g. p. register RS to special register RD.
read_cpb	1.8	34	Read capabilities register RS into g. p. register RD.
write_cpb	1.8	35	Write g. p. register RS to capabilities register RD.
read_perf	1.8	36	Read performance counter.
read_perfs	1.8	37	Read performance counter, serializing.

read_sys	1.8	38	Read system register RS into g. p. register RD.
write_sys	1.8	39	Write g. p. register RS to system register RD.
load_hi	2.6	0	Load 32-bit constant into the high part of a general purpose register. The low part is zero. RD = IM2 << 32.
insert_hi	2.6	1	Insert 32-bit constant into the high part of a general purpose register, leaving the low part unchanged. RD = (RS & 0xFFFFFFFF)	(IM2 << 32).
add_unsigned	2.6	2	Add zero-extended 32-bit constant to general purpose register.
sub_unsigned	2.6	3	Subtract zero-extended 32-bit constant from general purpose register.
add_hi	2.6	4	Add 32-bit constant to high part of general purpose register. RD = RS + (IM2 << 32).
and_hi	2.6	5	AND high part of general purpose register with 32-bit constant. RD = RS & (IM2 << 32).
or_hi	2.6	6	OR high part of general purpose register with 32-bit constant. RD = RS	(IM2 << 32).
xor_hi	2.6	7	XOR high part of general purpose register with 32-bit constant. RD = RS ^ (IM2 << 32).
address	2.6	32	RD = RS + IM2, RS can be THREADP (28), DATAP (29) or IP (30).

Table 4.9: List of single-format instructions with vector registers and mixed register types

Instruction	Format	OP1, OP2	Description
set_len	1.2	0	RD = vector register RT with length changed to value of RS.
get_len	1.2	1	Get length of vector register RS into general purpose register RD.
set_num	1.2	2	Change the length of vector register to RS·OS.
get_num	1.2	3	Get length of vector register divided by the operand size.
compress	1.2	4	Compress vector RT of length RS to a vector of half the length and half the element size. Double precision → single precision, 64-bit integer → 32-bit integer, etc.
compress_ss	1.2	5	Compress integer vector RT of length RS to a vector of half the length and half the element size, signed with saturation (optional).
compress_us	1.2	6	Compress integer vector RT of length RS to a vector of half the length and half element size, unsigned with saturation (optional).

36

expand	1.2	7	Expand vector RT of length RS/2 and half the specified element size to a vector of length RS with the specified element size. Half precision $\rightarrow$ single precision, 32-bit integer $\rightarrow$ 64-bit integer with sign extension, etc.
expand_us	1.2	8	Expand integer vector RT of length RS/2 and half the specified element size to a vector of length RS with the specified element size. 32-bit integer $\rightarrow$ 64-bit integer with zero extension, etc.
compress_sparse	1.2	9	Compress sparse vector elements indicated by mask bits into contiguous vector. RS = length of input vector. (optional).
expand_sparse	1.2	10	Expand contiguous vector into sparse vector with positions indicated by mask bits. RS = length of output vector. (optional).
extract	1.2	11	Extract one element from vector RT, starting at offset RS·OS, with size OS into scalar in vector register RD.
insert	1.2	12	Replace one element in vector RD, starting at offset RS·OS, with scalar RT.
broadcast	1.2	13	Broadcast first element of vector RT into all elements of RD with length RS.
bits2bool	1.2	14	The lower n bits of RT are unpacked into a boolean vector RD with length RS, with one bit in each element, where n = RS / OS.
bool2bits	1.2	15	The boolean vector RT with length RS is packed into the lower n bits of RD, taking bit 0 of each element, where n = RS / OS. The length of RD is at least sufficient to contain n bits.
bool_reduce	1.2	16	The boolean vector RT with length RS is reduced by combining bit 0 of all elements. The output is a scalar integer where bit 0 is the AND combination of all the bits, and bit 1 is the OR combination of all the bits. The remaining bits are reserved for future use.
shift_expand	1.2	18	Shift vector RT up by RS bytes and extend the vector length by RS. The lower RS bytes of RD will be zero.
shift_reduce	1.2	19	Shift vector RT down RS bytes and reduce the length by RS. The lower RS bytes of RT are lost.
shift_up	1.2	20	Shift elements of vector RT up RS elements. The lower RS elements of RD will be zero, the upper RS elements of RT are lost.
shift_down	1.2	21	Shift elements of vector RT down RS elements. The upper RS elements of RD will be zero, the lower RS elements of RT are lost.

div_ex_s	1.2	24	Divide vector of double-size signed integers RS by signed integers RT. RS has element size 2·OS. These are divided by the even numbered elements of RT with size OS. The results are stored in the even-numbered elements of RD. The remainders are stored in the odd-numbered elements of RD. (Optional for vectors).
div_ex_u	1.2	25	Same, with unsigned integers. (Optional for vectors).
sqrt	1.2	26	Square root (floating point, optional).
add_c	1.2	28	Add with carry. Vector has two elements. The upper element is used as carry on input and output (optional).
sub_b	1.2	29	Subtract with borrow. Vector has two elements. The upper element is used as borrow on input and output (optional).
add_ss	1.2	30	Add integer vectors, signed with saturation (optional).
add_us	1.2	31	Add integer vectors, unsigned with saturation (optional).
sub_ss	1.2	32	Subtract integer vectors, signed with saturation (optional).
sub_us	1.2	33	Subtract integer vectors, unsigned with saturation (optional).
mul_ss	1.2	34	Multiply integer vectors, signed with saturation (optional).
mul_us	1.2	35	Multiply integer vectors, unsigned with saturation (optional).
shl_ss	1.2	36	Shift left integer vectors, signed with saturation (optional).
shl_us	1.2	37	Shift left integer vectors, unsigned with saturation (optional).
add_oc	1.2	38	add with overflow check (optional).
sub_oc	1.2	39	subtract with overflow check (optional).
subr_oc	1.2	40	subtract reverse with overflow check (optional).
mul_oc	1.2	41	multiply with overflow check (optional).
div_oc	1.2	42	divide with overflow check (optional).
input	1.2	48	read from input port. RD = vector register, RT = port address, RS = vector length (privileged instruction).
output	1.2	49	write to output port. RD = vector register source operand, RT = port address, RS = vector length (privileged instruction).
gp2vec	1.3 B	0	Move value of general purpose register RS to scalar in vector register RD.
set_bits_x	1.3 B	1	Set all bits except one. RD = RS | ˜ (1 << IM1).
clear_bits_x	1.3 B	2	Clear all bits except one. RD = RS & (1 << IM1).
make_sequence	1.3 B	3	Make a vector with RS sequential numbers. First value is IM1.

mask_length	1.3 B	4	Make mask with true in the first RS bytes. Option bits in IM1.
vec2gp	1.3 B	8	Move value of first element of vector register RS to general purpose register RD.
bitscan_f	1.3 B	9	Bit scan forward. Find index to lowest set bit in RS (optional for vectors).
bitscan_r	1.3 B	10	Bit scan reverse. Find index to highest set bit in RS (optional for vectors).
float2int	1.3 B	12	Conversion of floating point to integer with the same operand size. The rounding mode is specified in IM1.
int2float	1.3 B	13	Conversion of integer to floating point with same operand size.
round	1.3 B	14	Round floating point to integer in floating point representation. The rounding mode is specified in IM1.
round2n	1.3 B	15	Round to nearest multiple of 2^n. $RD = 2^n \cdot \text{round}(2^{-n} \cdot RS)$. n is a signed integer constant in IM1 (optional).
abs	1.3 B	16	Absolute value of integer. Uses saturation if $IM1 = 1$.
popcount	1.3 B	17	Count the number of bits in RS that are 1.
broadcast	1.3 B	18	Broadcast 8-bit constant into all elements of RD with length RS (31 in RS field gives scalar output).
fp_category	1.3 B	19	Check if floating point numbers belong to the categories indicated by constant.
byte_reverse	1.3 B	20	Reverse the order of bytes in each element of vector.
bit_reverse	1.3 B	21	Reverse the order of bits in each element of vector (optional).
truth_tab2	1.3 B	24	Boolean function of two inputs, given by a truth table.
read_spev	1.3 B	30	Read special register RT into vector register RD with length RS.
move	1.3 C	32	Move 16 bit constant to 16-bit scalar (optional).
add	1.3 C	33	Add broadcast 16 bit constant to 16-bit vector elements (optional).
and	1.3 C	34	AND broadcast 16 bit constant with 16-bit vector elements (optional).
or	1.3 C	35	OR broadcast 16 bit constant with 16-bit vector elements (optional).
xor	1.3 C	36	XOR broadcast 16 bit constant with 16-bit vector elements (optional).
move	1.3 C	38	$RD = IM2 << IM1$. Sign-extend IM2 to 32 bits and shift left by the unsigned value IM1 to make 32 bit scalar (optional).
move	1.3 C	39	$RD = IM2 << IM1$. Sign-extend IM2 to 64 bits and shift left by the unsigned value IM1 to make 64 bit scalar (optional).

add	1.3 C	40	RD += IM2 << IM1. Add broadcast shifted signed constant to 32-bit vector elements (optional).
add	1.3 C	41	RD += IM2 << IM1. Add broadcast shifted signed constant to 64-bit vector elements (optional).
and	1.3 C	42	RD &= IM2 << IM1. AND broadcast shifted signed constant with 32-bit vector elements (optional).
and	1.3 C	43	RD &= IM2 << IM1. AND broadcast shifted signed constant with 64-bit vector elements (optional).
or	1.3 C	44	RD |= IM2 << IM1. OR broadcast shifted signed constant with 32-bit vector elements (optional).
or	1.3 C	45	RD |= IM2 << IM1. OR broadcast shifted signed constant with 64-bit vector elements (optional).
xor	1.3 C	46	RD ^= IM2 << IM1. XOR broadcast shifted signed constant with 32-bit vector elements (optional).
xor	1.3 C	47	RD ^= IM2 << IM1. XOR broadcast shifted signed constant with 64-bit vector elements (optional).
add	1.3 C	48	RD += IM21 << 16. Add broadcast signed 16-bit constant shifted left by 16 to 32-bit vector elements (optional).
add	1.3 C	49	RD += IM21 << 16. Add broadcast signed 16-bit constant shifted left by 16 to 64-bit vector elements (optional).
mov	1.3 C	56	Move converted half precision floating point constant to single precision scalar (optional).
mov	1.3 C	57	Move converted half precision floating point constant to double precision scalar (optional).
add	1.3 C	58	Add broadcast half precision floating point constant to single precision vector (optional).
add	1.3 C	59	Add broadcast half precision floating point constant to double precision vector (optional).
mul	1.3 C	60	Multiply broadcast half precision floating point constant with single precision vector (optional).
mul	1.3 C	61	Multiply broadcast half precision floating point constant with double precision vector (optional).
concatenate	2.5	2, 8	A vector RT of length RS and a vector RU of length RS are concatenated into a vector RD of length 2·RS.
permute	2.5	2, 9	The vector elements of RT are permuted within each block of size RS bytes, using indices in RU. Each index is relative to the beginning of a block. An index out of range produces zero. The maximum block size is implementation dependent.

interleave	2.5	2,10	Interleave elements of vectors RT and RU to produce vector RD of length RS. Even-numbered elements of the destination come from RT and odd-numbered elements from RU. (optional).
truth_tab3	2.5	3, 8	Boolean function of three inputs, given by a truth table (optional).
truth_tab4	2.5	4, 8	Boolean function of four inputs, given by a truth table (optional).
mul_add	2.5	3, 9	$RD = \pm RS \pm RT \cdot RU$ (optional but recommended).
add_add	2.5	3,10	$RD = \pm RS \pm RT \pm RU$ (optional).
add_add_add	2.5	3,11	$RD = \pm RS \pm RT \pm RU \pm IM2$ Add three vector register operands and a 16-bit constant IM2 (optional).
add_add_add	2.5	4,11	$RD = \pm RD \pm RS \pm RT \pm RU$ Add four vector register operands (optional).
repeat_block	2.5	8, 8	Repeat a block of data to make a longer vector. RT is input vector containing data block to repeat. RU is length of block to repeat (g. p. register). RS is length of destination vector RD. (optional. RU must be a multiple of 4).
repeat_within _blocks	2.5	8, 9	Broadcast the first element of each block of data in a vector to the entire block. RT is input vector containing data blocks. RU is length of each block (g. p. register). RS is length of destination vector RD. The operand size must be at least 4 bytes. (optional).
load_hi	2.7	16	Make vector of two elements. $dest[0] = 0$, $dest[1] = IM2$.
insert_hi	2.7	17	Make vector of two elements. $dest[0] = src1[0]$, $dest[1] = IM2$.
make_mask	2.7	18	Make vector where bit 0 of each element comes from bits in IM2, the remaining bits come from RS.
replace	2.7	19	Replace elements in RS by constant IM2.
replace_even	2.7	20	Replace even-numbered elements in RS by constant IM2.
replace_odd	2.7	21	Replace odd-numbered elements in RS by constant IM2.
broadcast	2.7	22	Broadcast 32-bit constant into all elements of RD with length RS (31 in RS field gives scalar output).
permute	2.7	33	The vector elements of RT are permuted within each block of size RS bytes. The $4 \cdot n$ bits of IM2 are used as index with 4 bits for each element in blocks of size n. The same pattern is used in each block. The number of elements in each block, $n = RS / OS \leq 8$.

Instruction	Format	OP1, OP2	Description
store	2.7 B	48	Store 32-bit constant IM2 to memory operand with base RT and 8-bit offset IM1 (optional).
fence	2.4.x	0, 8	Memory fence. read, write or full indicated by OP3.
cmp_swap	2.8.x	1, 8	Atomic compare and exchange.
read_insert	2.4.0 2.4.3	2, 8	Replace one element in vector RD, starting at offset RS·OS, with scalar memory operand (optional).
move_store	2.4.x	3, 8	Conditional move and store. Mask bits = 01 or 11: store RU. Mask bits = 10: store zero. Mask bits = 11: store RD. (optional).
extract_store	2.4.0	3, 9	Extract one element from vector RD, starting at offset RS·OS, with size OS into memory operand with base RT and offset IM2 (optional).
extract_store	2.4.3	3, 9	Extract one element from vector RD, starting at offset RS·OS, with size OS into memory operand with base RT, scaled index RU and unsigned limit RU ≤ IM2 (optional).
compress_store	2.4.1	3, 10	Compress vector RD of length RS to a vector of half the length and half the element size. Double precision → single precision, 64-bit integer → 32-bit integer, etc. Store at memory with base RT, offset IM2, length RS/2 (optional).
add_store	2.4.x	4, 8	Add RD and RU, store the result to memory operand (optional).
sub_store	2.4.x	4, 9	Subtract RU from RD, store the result to memory operand (optional).
mul_store	2.4.x	4, 10	Multiply RD and RU, store the result to memory operand (optional).
read_memory_map	2.4.2	48, 8	Read memory map. RD = map entry, RT = memory pointer, RS = vector length and negative index to both source and destination (privileged).
write_memory_map	2.4.2	48, 9	Write memory map. RD = map entry, RT = memory pointer, RS = vector length and negative index to both source and destination (privileged).

4.4 Description of instructions

Instructions that need special explanation are described in this section.

Multi-format instructions

nop

It is recommended to code NOPs as 32-bit words of all zeroes. The processor is allowed to skip this type of NOPs as fast as it can at an early stage in the pipeline. A pair of tiny instructions where the second instruction is a NOP can be treated as a single instruction.

These NOPs cannot be used as timing delays, only as fillers.

move

Copy value from a register, memory operand or immediate constant to a register. If the destination is a vector register and the source is an immediate constant then the result will be a scalar. The value will not be broadcast because there is no other input operand that specifies the vector length. If a vector is desired then use the broadcast instruction instead.

The move instruction with an immediate operand is the preferred method for setting a register to zero.

The move instruction has several additional tiny and single-format variants. The assembler will normally choose the shortest variant that fits the specified operands.

store

The source and destination operands are swapped so that the value of RD is written to a memory operand. Only formats that specify a memory operand (scalar or vector without broadcast) are allowed.

The size of the memory operand is determined by the operand size OS when a scalar memory operand is specified, or by the vector length register in RS when a vector memory operand is specified.

The hardware must be able to handle memory operand sizes that are not powers of 2 without touching additional memory (read and rewrite beyond the memory operand is not allowed unless access from other threads is blocked during the operation and any access violation is suppressed). It is allowed to write the operand in a piecemeal fashion.

Masked operation with bit 0 and 1 both zero will write zero to the memory.

Masked operation with bit $0 = 0$ and bit $1 = 1$ may or may not be supported for vector registers. If supported, this combination will leave the memory position untouched. This cannot be implemented as read-combine-write because this would not be thread-safe.

prefetch

Prefetch memory operand into cache. Different variants can be specified by bit 0-3 of OP3 for format 2.4 and 2.8.

sign_extend

The input can be an 8-bit, 16-bit or 32-bit integer. This integer is sign-extended to produce a 64-bit output in a general purpose register or a scalar in a vector register. If the input is a vector then only the first element in each 64-bit block of the input vector is used. Floating point types cannot be used.

min and max

min(src1,src2) = src1 < src2 ? src1 : src2
max(src1,src2) = src1 < src2 ? src1 : src2

The operands are treated as signed. There is also a version for unsigned integers:

min_u(src1,src2) = src1 < src2 ? src1 : src2
max_u(src1,src2) = src1 < src2 ? src1 : src2

When the unsigned version is applied to floating point operands, it takes the absolute values of the operands, and the instruction name is changed:

min_abs(src1, src2) = min(abs(src1), abs(src2))
max_abs(src1, src2) = max(abs(src1), abs(src2))

The handling of floating point NAN operands is determined by bit 22 of the mask register or the mumeric control register. If bit 22 is zero then the non-nan operand is output when one of the inputs is NAN, in accordance with the IEEE Standard 754-2008. If bit 22 is one then the NAN input is propagated.

A NAN operand that is not propagated will generate a trap if flag bit 29 is set.

Bitwise boolean instructions

These instructions include: and, and_not, or, xor. Floating point operands are handled in the same way as integer operands.

Bit manipulation instructions

The following instructions are provided for manipulating bits:

extract_b: Extract bit number src2 in src1
set_b: Change bit number src2 in src1 to 1
clear_b: Change bit number src2 in src1 to 0
toggle_b: Change bit number src2 in src1 to its opposite

A floating point operand in src1 is treated as an integer with the same size. The bit index in src2 is interpreted as an 8-bit unsigned integer regardless of the operand type.

These instructions can be implemented with an 8-bit immediate constant for src2 instead of the larger constant that would be needed if we used AND, OR, XOR instructions for manipulating single bits. These instructions can also be used with floating point numbers, mainly for manipulating the sign bit.

mul_add

Fused multiply and add.

dest = ± src1 ± (src2 · src3)

The fused multiply-and-add instruction can often improve the performance of floating point code significantly.

Only instruction formats that allow three operands are supported.

The signs of the operands can be inverted as indicated by bits 0-3 of the OP3 field in formats that use the E2 template, including the extra format 2.5, with:

bit 0: change sign of src1 in even-numbered vector elements
bit 1: change sign of src1 in odd-numbered vector elements
bit 2: change sign of src2·src3 in even-numbered vector elements
bit 3: change sign of src2·src3 in odd-numbered vector elements

This makes it possible to do multiply-and-add, multiply-and-subtract, multiply-and-reverse-subtract, etc. It can also do multiply with alternating add and subtract, which is useful in calculations with complex numbers. There is no sign change in other formats where the OP3 field is absent. An additional single-format version of mul_add is supplied with four register operands and an OP3 field.

The OP3 field is not used as shift count in formats 2.5 and 2.9.

Support for integer operands is optional. Support for floating point operands is optional but desired.

add_add

Two additions in one instruction.

dest = ± src1 ± src2 ± src3

Only instruction formats that allow three operands are supported.

The signs of the operands can be inverted as indicated by bits 0-2 of the OP3 field in formats that use the E2 template, including the extra format 2.5:

bit 0: change sign of src1
bit 1: change sign of src2
bit 2: change sign of src3

There is no sign change in other formats where the OP3 field is absent. An additional single-format version of add_add is supplied with four register operands and an OP3 field.

The OP3 field is not used as shift count in formats 2.5 and 2.9.

The precision for floating point operands is preferably better than the least significant bit of the numerically highest operand, but the intermediate result is not calculated with unlimited precision. The hardware implementation can adjust the exponents of all operands in the first clock cycle and use the adder network of the multiplication circuit.

This instruction should only be supported if it can be implemented so that it is faster than two consecutive add instructions. It may be supported for integer operands or floating point or both. See also add_add_add page 55.

Compare instructions

A compare instruction compares two source operands and stores the result in bit 0 of the destination. The condition is determined by an additional code stored in the third source operand when formats 0.0-0.3 or 2.0-2.3 are used. Formats that use the E2 template (2.4, 2.5, 2.8, 2.9) are coded differently: The condition code is in the OP3 field. The 16-bit IM2 field in the formats 2.5 and 2.9 is used as the second source operand. This operand is not shifted by OP3.

The remaining bits of the result are copied from the mask register, or from the numeric control word if no mask is used. This is suitable when the result is used as a mask.

The condition code is defined in this table:

Table 4.11: Condition codes for compare instruction

Bit	Meaning
0	Inverts the condition.
1-2	Determines the condition: 0 = smaller, 1 = equal, 2 = bigger, 3 = unordered.
3	For integer operands: 0 = signed operands, 1 = unsigned operands. For floating point operands: This bit indicates the result if one or both operands are NAN.

Compare instructions can be masked. Bit 0 of the result is equal to bit 1 of the mask register if bit 0 of the mask register is zero.

Tiny format instructions

clear

This instruction sets the length of a vector register to zero. All contents is lost. The register can then be regarded as unused.

Push and pop operations

There are no push and pop instructions. A general purpose register R can be pushed on the stack with the following pair of tiny instructions:

```
add  sp,−8
store  [sp],R
```

A general purpose register R can be popped from the stack with the following pair of tiny instructions:

```
move  R,[sp]
sub  sp,−8
```

Note that the constant -8 can be contained in the 4-bit signed field RS, but the constant 8 cannot. This is the reason why we are adding and subtracting -8 rather than doing the opposite with +8.

The assembler may support macros named push and pop for these sequences.

Saving and restoring vector registers

When saving a vector register with variable length, we do not want to save the maximum length when only part of the register is used. Therefore, we have the save_cp and restore_cp instructions which are intended for saving and restoring a vector register without using more memory than necessary.

Note that the format for the saved image is implementation-dependent. Typically, the save_cp instruction will save the length of the vector followed by as many bytes as indicated by the length, and the restore_cp instruction will read the length and then read as many bytes as indicated by the length.

The microprocessor is allowed to compress the data in any way that it can handle sufficiently fast. For example, a boolean vector that uses only one bit per element can obviously be compressed to a much smaller size. The image for an unused vector register will typically contain only a few bytes of zero for the length.

The software should never use the saved image for anything else than restoring a vector register on the same microprocessor model that saved it, because the image format is not compatible across microprocessors.

The size of the saved image can be added to a pointer with the add_cps instruction or subtracted from a pointer with the sub_cps instruction. RS indicates the pointer, which can be r0-14 or r31 (stack pointer).

A vector register V can be saved (pushed) on the stack with the following pair of tiny instructions:

```
sub_cps  sp,V
save_cp  [sp],V
```

A vector register V can be restored (popped) from the stack with the following pair of tiny instructions:

```
restore_cp  V,[sp]
add_cps  sp,V
```

The same instructions can be used for saving vector registers during a task switch. Unused vector registers will only use very little space when saved in this way.

The size of the compressed image, as indicated by the add_cps and sub_cps instructions, must be a multiple of 8 when the stack pointer is used in order to keep the stack properly aligned.

It is allowed to use a smaller size that is not a multiple of 8 during a task switch where, typically, another pointer register is used. In this case, a control register must be provided to control the format of the saved image.

The restore_cp instruction is allowed to read more bytes than necessary, up to the maximum vector length plus 8 bytes, and discard any superfluous bytes afterwards when the actual length is known.

Single-format instructions that use general purpose registers and special registers

read_spe, write_spe

Read or write a special register. The following special registers are currently defined. The size is 64 bits. These registers are initialized with their default values at program start.

Table 4.12: List of special registers

Special register number	Meaning
0	Numeric control register (NUMCONTR)
1	Microprocessor brand ID
2	Microprocessor version number
28	Thread environment block pointer (THREADP)
29	Data section pointer (DATAP)

read_cpb, write_cpb

Read or write processor capabilities register. These registers are used for indicating capabilities of the processor, such as support for optional instructions and limitations to vector lengths. The size is 64 bits. These registers are initialized with their default values at program start.

The immediate constant in IM1 determines details of the operation:

Table 4.13: Meaning of immediate constant in read_cpb and write_cpb instructions

Bit number	Meaning
0	0: read/write the capabilities for the operand type specified in bit 5-7. 1: read the typical capabilities for all operand types / write the capabilities for all relevant operand types.
1	0: read the current value of the register, which may have been modified. 1: read the real capabilities of the hardware (cannot write.)
5-7	Operand type for capabilities.

Table 4.14: List of capabilities registers

Capabilities register number	Meaning

0	Maximum vector length for general instructions.
1	Maximum vector length for permute instructions.
2	Maximum block size for permute instructions.
3	Maximum vector length for compress_sparse and expand_sparse.
8	Support for optional instructions in general purpose registers. Each bit indicates a specific instruction.
9	Support for optional instructions on scalars in vector registers. Each bit indicates a specific instruction.
10	Support for optional instructions on vectors. Each bit indicates a specific instruction.

Changing the values of the maximum vector length has the following effects. If the maximum length is reduced below the physical capability then any attempt to make a longer vector will result in the reduced length. The behavior of vector registers that already had a longer length before the maximum length was reduced, is implementation dependent. If the maximum vector length is set to a higher value than the physical capability then any attempt to make a vector longer than the physical capability will cause a trap to facilitate emulation. Capabilities registers 0-3 can be increased for the purpose of emulation. The value of capabilities registers 0-3 must be powers of 2.

Capabilities registers 8-9 can be modified for test purposes or to tell the software not to use a specific instruction. The same value will be returned when reading the register. Attempts to execute an instruction that is not supported will cause a trap, regardless of the value of the capabilities register.

read_sys, write_sys

These instructions are for accessing various registers that are only accessible in mode.

read_perf

Read the internal clock count, number of instructions executed, or other performance-related counts.

read_perfs

Same as read_perf. This instruction is serializing, which means that it cannot execute out of order.

popcount

The popcount instruction counts the number of 1-bits in an integer. It can also be used for parity generation.

bitscan_f

Bit scan forward.

Find index to lowest set bit, i. e. highest X for which
$(((1 << X) - 1) \& src1)) == 0$.

bitscan_r

Bit scan reverse.
Find index to highest set bit, i. e. highest X for which $(1 << X) \leq src1$.

round_d2

Round down to nearest power of 2, i. e. $1 << bit\_scan\_reverse(src1)$.

round_u2

Round up to nearest power of 2, i. e.
(S & (S-1)) == 0 ? S : 1 << (bit_scan_reverse(S) + 1),
where S = src1.

shift_add

Shift and add. dest = src1 + (src2 << src3).

src1 uses the same register as dest. src3 is an 8-bit signed immediate constant.

Will shift right with zero extension if src3 is negative.

address

Calculate an address relative to a pointer by adding a 32-bit sign-extended constant to a general purpose register or a special register. The pointer register can be r0-r27, THREADP (28), DATAP (29), IP (30) or SP(31).

cmp_swap

Atomic compare and swap instruction, used for thread synchronization and for lock-free data sharing between threads. src1 and src2 are register operands, src3 is a memory operand, which must be aligned to a natural address. All operands are treated as integers, regardless of the specified operand type. The operation is:

```
temp = src3;
if (temp == src1) src3 = src2;
return temp;
```

Further atomic instructions can be implemented, if needed, in format 2.8 with OP1 = 1 and increasing values of OP2.

Single-format instructions with g. p. register input and vector output, or vice versa

gp2vec

The value of a general purpose register is copied to a scalar in a vector register. The length will be the operand size. No type conversion is made.

vec2gp

The first element of a vector register is copied to a general purpose register. If an integer type less than 64 bits is specified then the value is sign-extended to 64 bits. If a single-precision float type is specified then the value is zero-extended to 64 bits. No other type conversion is made.

set_len

Sets the length of a vector register to the number of bytes specified by a general purpose register. If the specified length is more than the maximum length for the specified operand type then the maximum length will be used.

If the output vector is longer than the input vector then the extra elements will be zero. If the output vector is shorter than the input vector then the extra elements will be discarded.

get_len

Gets the length of a vector register in bytes. The result is stored in a general purpose register.

set_num

Same as set_len, the length is multiplied by the operand size.

get_num

Same as get_len, the length is divided by the operand size.

mask_length

Make a boolean vector to mask the first n elements of a vector, where
n = RS / (operand size). The output vector RD will have the same length as the input vector RD.
RS indicates the length of the part that is enabled by the mask. IM1 contains the following option bits:
bit 0 = 0: bit 0 will be 1 in the first n elements in the output and 0 in the rest.
bit 0 = 1: bit 0 will be 0 in the first n elements in the output and 1 in the rest.
bit 1 = 1: set bit 1 of all elements in the output to 1.
bit 2 = 1: copy bit 1 of each element from input vector RD.
bit 3 = 1: copy bit 1 of each element from the numeric control register.
bit 4 = 1: copy remaining bits from input vector RD.
bit 5 = 1: copy remaining bits from the numeric control register.
Output bits that are not set by any of these options will be zero.

make_sequence

Makes a vector of length RS bytes. The number of elements is RS/(operand size). The first element is equal to IM1, the next element is IM1+1, etc. Support for floating point is optional.

Other single-format instructions that may change the length of a vector

bits2bool

Expand contiguous bits in a vector register to a boolean vector with one bit in each element.

bool2bits

Convert a boolean vector of n elements to n contiguous bits in a vector register. The length of the destination vector will be a power of 2 sufficient to hold n bits.

shift_expand

The length of a vector is expanded by the specified number of bytes by adding zero-bytes at the low end and shifting all bytes up. If the resulting length is more than the maximum vector length for the specified operand type then the upper bytes are lost.

shift_reduce

The length of a vector is reduced by the specified number of bytes by removing bytes at the low end and shifting all bytes down. If the resulting length is less than zero then the result will be a zero-length vector. The specified operand type is ignored.

compress

The elements of a vector are converted to half the element size. The length of the output vector will be half the length of the input vector. The OT field specifies the operand type of the input vector.

Double precision floating point numbers are converted to single precision. Integer elements are converted to half the size by discarding the upper bits. Support for the following conversions are optional: single precision float to half precision, quadruple precision to double precision, 8-bit integer to 4-bit.

If the length of the input vector differs from the length specified by RS, then the length is converted to RS before compression.

compress_ss

Same as compress. Integers are treated as signed and compressed with saturation. Floating point operands cannot be used. This instruction is optional.

compress_us

Same as compress. Integers are treated as unsigned and compressed with saturation. Floating point operands cannot be used. This instruction is optional.

expand

This is the opposite of compress. The output vector has the specified length and the input vector has half this length. The OT field specifies the operand type of the output vector. Single precision floating point numbers are converted to double precision. Integers are converted to the double size by sign-extension. Support for the following conversions are optional: half precision float to single precision, double precision to quadruple precision, 4-bit integer to 8-bit.

If the length of the input vector differs from RS/2 then the length is converted before expansion. If the resulting length exceeds the maximum vector length for the specified operand type then the extra elements are lost.

expand_us

Same as expand. Integers are expanded by zero-extension. Floating point operands cannot be used.

Single-format instructions that can move data horizontally within vector

The latency of these instructions may depend on the distance of moving (specified by RS) for very long vectors.

extract

Extract one element of a vector into a scalar in a vector register. An index out of range will produce zero. An operand size of 16 bytes can be used, even if this size is not otherwise supported.

insert

Replace one element of a vector by inserting a scalar into the position indicated by the index. An index out of range will leave the vector unchanged. An operand size of 16 bytes can be used, even if this size is not otherwise supported.

interleave

Interleave the inputs from two vectors so that the even-numbered elements come from the first input vector and the odd-numbered elements come from the second input vector. Optional.

shift_up

Shift elements of a vector up by the number of elements indicated by RS. The lower RS elements of the output will be zero, the upper RS elements of the input are lost.

This instruction differs from shift_expand by indicating the shift count as a number of elements rather than a number of bytes, and by not changing the length of the vector.

shift_down

Shift elements of a vector down by the number of elements indicated by RS. The upper RS elements of the output will be zero, the lower RS elements of the input are lost.

This instruction differs from shift_reduce by indicating the shift count as a number of elements rather than a number of bytes, and by not changing the length of the vector.

permute

This instruction permutes the elements of a vector. The vector is divided into blocks of size RS bytes each. The block size must be a power of 2 and a multiple of the operand size. Elements can be moved arbitrarily between positions within each block, but not between blocks. Each element of the output vector is a copy of an element in the input vector, selected by the corresponding index in an index vector. The indexes are relative to the start of the block they belong to, so that an index of zero will select the first element in the block of the input vector and insert it in the corresponding position of the output vector. The same element in the input vector can be copied to multiple elements in the output vector. An index out of range will produce a zero. The indexes are interpreted as an integers regardless of the operand type.

The permute instruction has two versions. The first version specifies the indexes in a vector with the same length and element size as the input vector.

The second version specifies the indexes as a 32-bit immediate constant with 4 bits per element. This constant is split into a maximum of 8 elements with 4 bits in each. If the blocks have more than 8 elements each then the sequence of 8 elements is repeated to fill a block. The same pattern of indexes will be applied to all blocks in this version of the permute instruction.

The maximum block size for the permute instruction is implementation-dependent and given by a special register. The reason for this limitation of block size is that the complexity of the hardware grows quadratically with the block size. A full permutation is possible if the vector length does not exceed the maximum block size. A trap is generated if RS is bigger than the maximum block size.

There are two ways to combine the outputs of multiple permute instructions. One method is to use indexes out of range to produce zeroes for unused outputs and then OR'ing the outputs. Another method is to use masks to combine the outputs.

Permute instructions are useful for reordering data, for transposing a matrix, etc.

Permute instructions can also be used for parallel table lookup when the block size is big enough to contain the entire table.

Finally, permute instructions can be used for gathering and scattering data within an area not bigger than the vector length or the block size.

broadcast

Copies the first element of the input vector to all elements of the output vector. An element size of 16 bytes (128 bits) is supported if the maximum vector length is more than 16 bytes, even if this size is not otherwise supported.

repeat_block

This does the same as broadcast, but with larger blocks of data. The block size, which must be a multiple of 4, is given by a general purpose register. There may be a maximum limit to the block size. The hardware implementation may read the block size register in the address-calculation stage in the pipeline. This instruction is useful in matrix multiplication. Optional.

repeat_within_blocks

This divides a vector into blocks and broadcasts the first element of each block to the rest of the block. For example, if the input vector contains (0,1,2,3,4,5,6,7,8) and the block size is 3 times the operand size, then the result will be (0,0,0,3,3,3,6,6,6). The operand size must be at least 4 bytes, and the block size must be a multiple of the operand size. The block size is given by a general purpose register. There may be a maximum limit to the block size. The hardware implementation may read the block size register in the address-calculation stage in the pipeline. This instruction is useful in matrix multiplication. Optional.

Other single-format vector instructions

Saturated arithmetic

add_ss, add_us, sub_ss, sub_us, mul_ss, mul_us, shl_ss, shl_us.

These instructions are used for arithmetic operations with saturation. An overflow will result in the maximum value for the given operand size. An underflow will result in the minimum value.

Support for these instructions is optional.

Add with carry and subtract with borrow

add_c, sub_b

dest and src1 are vectors of two integers. src2 is a vector of integers, where only the first element is used.

add_c:

```
sum = src1[0] + src2[0] + (src1[1] & 1)
dest[0] = bit 0−63 of sum
dest[1] = bit 64 of sum
```

sub_b:

```
sum = src1[0] − src2[0] − (src1[1] & 1)
dest[0] = bit 0−63 of sum
dest[1] = bit 64 of sum
```

Support for these instructions is optional. Longer vectors are not supported. See page 57 for an alternative for longer vectors.

Arithmetic instructions with overflow check

add_oc, sub_oc, subr_oc, mul_oc, div_oc.

These instructions use the even-numbered vector elements for arithmetic instructions. Each following odd-numbered vector element is used for overflow detection. If the first source operand is a scalar then the result operand will be a vector with two elements.

Overflow conditions are indicated with the following bits:

bit 0. Unsigned integer overflow (carry).
bit 1. Signed integer overflow.

bit 2. Floating point overflow.

bit 3. Floating point invalid operation.

The values are propagated so that the overflow result of the operation is OR'ed with the corresponding values of both input operands.

These instructions are optional.

Extended division

div_ex_s, div_ex_u

These instructions are optional. They may be supported for both scalars and vectors, for scalars only, or not at all.

byte_reverse

This instruction reverses the order of bytes in an integer. It can be used when reading and writing binary data files with big endian data organization.

read_spev

The value of the RT field indicates a special register to read. The output is a vector register with length specified by RS.

The following special registers are currently defined:

Table 4.15: Special registers that can be read into vectors

Special register number	Meaning
0	Numeric control register (NUMCONTR). The value is broadcast into all elements of the destinationregister with the indicated operand size and length.
1	Name of processor. The output is a zero-terminated UTF-8 string containing the brandname and model name of the microprocessor.

replace

All elements of src1 are replaced by the integer or floating point constant src2.

When used without a mask, the constant is simply broadcast to make a vector of the same length as src1. When used with a mask, the elements of src1 are selectively replaced. Elements that are not selected by the mask will be zero or unchanged, depending on bit 1 in the mask.

make_mask

Make a mask from the bits of the 32-bit integer constant src2. Each bit of src2 goes into bit 0 of one element of the output. The remaining bits of each element are taken from src1. The length of the output is the same as the length of src1. If there are more than 32 elements in the vector then the bit pattern of src2 is repeated.

fp_category

The input is a floating point vector. The output is a boolean vector indicating if the input belongs to any of the categories indicated by the bits in the immediate operand:

Table 4.16: Meaning of bits in fp_category

Bit number	Meaning
0	Invert result
1	Zero
2	Subnormal
3	Normal
4	Infinite
5	NAN
6	Sign bit
7	Copy remaining bits from mask or numeric control register.

Truth table functions

truth_tab2, truth_tab3, truth_tab4

These instructions can make an arbitrary boolean function of two, three or four boolean vector input variables, expressed by a truth table. The result in bit 0 of each vector element is the arbitrary boolean function of bit 0 of the corresponding elements of each of the input operands. Bit 0 of the output is a bit from the truth table selected by the combined input bits. The remaining bits of the output vector are copied from the mask register if there is one, or from the first input operand otherwise.

truth_tab2 has the inputs in RD and RS, the output in RD, and a 4-bit truth table in IM1.

truth_tab3 has the inputs in RS, RT and RU, the output in RD, and an 8-bit truth table in IM2.

truth_tab4 has the inputs in RD, RS, RT and RU, the output in RD, and a 16-bit truth table in IM2.

truth_tab4 must have an operand size of at least 16 bits. truth_tab3 and truth_tab4 are optional.

A mask can be used as an extra input operand for truth_tab3 and truth_tab4, according to the normal function of a mask.

These instructions can be used as universal instructions for manipulating and combining boolean vectors and masks.

The hardware implementation can use the existing barrel shifters, shifting the truth table right by the count defined by the combined bits of the input operands.

add_add_add

Adds four operands. The last operand can be a register operand or a 16-bit signed immediate operand. The signs of the operands can be inverted as indicated by bits 0-3 of the OP3 field:

bit 0: change sign of src1
bit 1: change sign of src2
bit 2: change sign of src3
bit 3: change sign of src4

See add_add page 45 for more details.

This instruction is optional.

4.5   Common operations that have no dedicated instruction

This section discusses some common operations that are not implemented as single instructions, and how to code these operations in software.

Change sign

For integer operands, do a reverse subtract from zero. For floating point operands, use the toggle_b instruction on the sign bit.

Floating point abs

To get the absolute value of a floating point number, use the clear_b instruction to clear the sign bit.

Not

To invert all bits in an integer, do an XOR with -1. To invert a Boolean, do an XOR with 1.

Rotate through carry

Rotates through carry are rarely used, and common implementations can be very inefficient. A rotate left through carry can be replaced by an add_c with the same register in both source operands.

Horizontal vector add

An instruction for adding all elements of a vector would be useful, but such an instruction is not supported because this would be a complex instruction with variable latency depending on the vector length.

The sum of all elements of a vector can be calculated by repeatedly adding the lower half and the upper half of the vector. This method is illustrated by the following example, finding the horizontal sum of a vector of 32-bit integers. The syntax for assembly language is described on page 89.

```
v0 = my_vector     // we want the horizontal sum of this vector
r0 = get_len(v0) // length of vector in bytes
r0 = roundu2.64(r0)  // round up to nearest power of 2
v0 = set_len(v0, r0) // adjust vector length
// Loop to calculate horizontal sum of v0
LOOP: // label
   // Half vector length
   r1 = shift_rightu.64(r0, 1)
   // Get upper half of vector
   v1 = shift_reduce(v0, r1)
   // Add upper half and lower half
   v0 = add.32(v1, v0)  // result has the length of the first operand
   // Half length for next iteration
   r0 = r1
   // loop while vector contains more than one element
   compare_unsign_jmpabove(r1, 4, LOOP)
// The sum is now a scalar in v0
```

The same method can be used for other horizontal operations. It may cause problems that the set_len instruction inserts elements of zero if the vector length is not a power of 2. Special care is needed if the operation does not allow extra elements of zero, for example if the operation involves multiplication or finding the minimum element. A possible solution is to mask off the unused elements in the first iteration. The following example finds the smallest element in a vector of floating point numbers:

```
v0 = my_vector                  // find the smallest element in this vector
r0 = get_len(v0)                // length of vector in bytes
r1 = roundu2.64(r0)             // round up to nearest power of 2
r1 = shift_rightu.64(r1, 1) // half length
v1 = shift_reduce(v0, r1)   // upper part of vector
r2 = sub.64(r0, r1)             // length of v1
```

```
    // use mask because the two operands may have different length
    v0 = set_len(v0, r1)          // reduce length of v0
    v2 = v0                       // arbitrary vector with length r1
    v2 = mask_length.32(v2, r2, 0x22)  // make mask for v1
    v0 = min.f(v0, v1, mask=v2)   // get minimum. mask off unused elements
    cmp_unsign_jmpbeloweq(r1, 4, ENDOFLOOP) // check if already finished
    // Loop to calculate horizontal minimum of v0
LOOP:  // label
       // Half vector length
       r2 = shift_rightu.64(r1, 1)
       // Get upper half of vector
       v1 = shift_reduce(v0, r2)
       // Get minimum of upper half and lower half
       v0 = min.f(v1, v0)    // result has the length of the first operand
       // Half length for next iteration
       r1 = r2
       // loop while vector contains more than one element
       compare_unsign_jmpabove(r2, 4, LOOP)
ENDOFLOOP:
    // The minimum is now a scalar in v0
```

High precision arithmetic

Function libraries for high precision arithmetic typically use a long sequence of add-with-carry instructions for adding integers with a very large number of bits. A more efficient method for big number calculation is to use vector addition and a carry-look-ahead method. The following algorithm calculates A + B, where A and B are big integers represented as two vectors of n·64 bits each, where n < 64.

```
v0 = A                       // first vector, n*64 bits
v1 = B                       // second vector, n*64 bits
v2 = carry_in                // scalar in vector register
v0 = add.64(v0, v1)          // sum without intermediate carries
v3 = compare.64(v0,v1,8)     // carry generate = (SUM < B). (unsigned compare)
v4 = compare.64(v0,−1,0xA)   // carry propagate = (SUM == −1)
v3 = bool2bits(v3)           // carry generate, compressed to bitfield
v4 = bool2bits(v4)           // carry propagate, compressed to bitfield
// CA = CP ^ (CP + (CG<<1) + CIN) // propagated additional carry
v3 = shift_left.64(v3,1)     // shift left carry generate
v2 = add.64(v2,v4)
v2 = add.64(v2,v3)
v2 = xor.64(v2,v4)
v1 = bits2bool(v2)           // expand additional carry to vector
v0 = sum.64(v0,v1)           // add correction to sum
r0 = get_num(v0)             // n = number of elements in vectors
v3 = gp2vec.64(r0)           // copy to vector register
v2 = shift_rightu.64(v2,v3)  // carry out
// v0 = sum, v2 = carry out
```

If the numbers A and B are longer than the maximum vector length then the algorithm is repeated. If the vector length is more than 64 * 8 bytes then the calculation of the additional carry involves more than 64 bits, which again requires a big number algorithm.

Matrix multiplication

Matrix operations can be difficult because they may involve a lot of permutations. The following example shows the multiplication of two 4×4 matrixes of floating point numbers, assuming that the vector registers are long enough to contain an entire matrix.

```
v1 = first_matrix          // first matrix, 4x4 floats
v2 = second_matrix         // second matrix, 4x4 floats
r0 = 4                     // loop counter
r1 = 16                    // row length in bytes
r2 = 64                    // size of entire matrix in bytes
r3 = 1                     // shift count, elements
r4 = 4                     // shift count, elements
v0 = broadcast.f(0,r2)     // make a matrix of zeroes
LOOP:                      // label
    v3 = repeat_within_blocks.f(v1,r1,r2) // repeat column
    v4 = repeat_block.f(v2,r1,r2)         // repeat row
    v0 = mul_add.f(v0,v3,v4)  // multiply rows and columns
    v1 = shift_down.f(v1,r3)  // next column
    v2 = shift_down.f(v2,r4)  // next row
    add_jmpnzero(r0,-1,LOOP)  // loop 4 times
// Result is in v0.
// You may roll out the loop and calculate partial sums separately to
// reduce the loop-carried dependency chain of v0
```

4.6   Unused instructions

Unused instructions and opcodes can be divided into three types:

1. The opcode is reserved for future use. Attempts to execute it will trigger a trap (synchronous interrupt) which can be used for generating an error message or for emulating instructions that are not supported.

2. The opcode is guaranteed to generate a trap, not only in the present version, but also in all future versions. This can be used as a filler in unused parts of the memory or for indicating unrecoverable errors. It can also be used for emulating user-specific instructions.

3. The error is ignored and does not trigger a trap. It can be used for future extensions that improve performance or functionality, but which can be safely ignored when not supported.

All three types are implemented, where type 1 is the most common.

Nop instructions with nonzero values in unused fields are type 3. These instructions are ignored.

Prefetch and fence instructions with no memory operand, with nonzero values in unused fields, or with undefined values in OP3 are type 3. These instructions are ignored.

Unused bits in masks and numeric control register are type 3. These bits are ignored.

Trap instructions and conditional trap instructions with nonzero values in unused fields or undefined values in any field are type 2. These instructions are guaranteed to generate a trap. A special version of the trap instruction is intended as filler in unused or inaccessible parts of code memory.

The undef instruction is type 2. It is guaranteed to generate a trap in all systems. It can be used for testing purposes and emulation.

The userdef__ instructions are type 1. These instructions are reserved for user-defined and application-specific purposes.

Instructions with erroneous coding should preferably behave as type 1. This includes instruction codes with nonzero values in unused fields, operand types not supported, or any other bit pattern with no defined meaning in any field. Type 3 behavior may alternatively be allowed in these cases. If so, the instruction should behave as if it were coded correctly.

All other opcodes not explicitly defined are type 1. These may be used for future instructions.

Small systems with no operating system and no trap support should define alternative behavior.

Chapter 5

Other implementation details

5.1 Endianness

The memory organization is little endian. Instructions for byte swapping are provided for reading and writing big endian binary data files.

Rationale

The storage of vectors in memory would depend on the element size if the organization was big endian. Assume, for example, that we have a 128 bit vector register containing four 32-bit integers, named A, B, C, D. With little endian organization, they are stored in memory in the order:

A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3,

where A0 is the least significant byte of A and D3 is the most significant byte of D. With big endian organization we would have:

A3, A2, A1, A0, B3, B2, B1, B0, C3, C2, C1, C0, D3, D2, D1, D0.

This order would change if the same vector register is organized, for example, as eight integers of 16 bits each or two integers of 64 bits each. In other words, we would need different read and write instructions for different vector organizations.

Little endian organization is more common for a number of reasons that have been discussed many times elsewhere.

5.2 Implementation of call stack

There are various methods for saving the return addresses for function calls: a link register, a separate call stack, or a unified stack for return addresses and local data. Here, we will discuss the pro's and con's of each of these methods.

Link register

Some systems use a link register to hold the return address. The advantage of a link register is that a leaf function can be called without storing anything on the stack. This saves cache bandwidth in programs with many leaf function calls. The disadvantage is that every non-leaf function needs to save the link register on a stack before calling another function, and restore the leaf register before returning.

If we decide to have a link register then it should be a special register, not one of the general purpose registers. A link register does not need to support all the things that a general purpose register can do. If the link register is included as one of the general purpose registers then it will be tempting for a programmer to save it to another register rather than on the stack, and then end the function by

jumping to that other register. This will work, of course, but it will interfere with the way returns are predicted. The branch predictor uses a special mechanism for predicting returns, which is different from the mechanism used for predicting other jumps and branches. This mechanism, which is called a return stack buffer, is a small rolling cache that remembers the addresses of the last calls. If a function returns by a jump to another register than the link register then it will use the wrong prediction mechanism, and this will cause severe delays due to misprediction of the subsequent series of returns. The return stack buffer will also be messed up if the link register is used for indirect jumps or other purposes.

The only instructions that are needed for the link register other than call and return, are push and pop. We can reduce the number of instructions in non-leaf functions by making a combined instruction for "push link register and then call a function" which can be used for the first function call in a non-leaf function, and another instruction for "pop link register and then return" to end a non-leaf function. However, this will violate the principle that we want to avoid complex instructions in order to simplify the pipeline design.

The only performance gain we get from using a link register is that it saves cache bandwidth by not saving the return address on leaf function calls. It will not affect performance in applications where cache bandwidth is not a bottleneck. The performance of the return instruction is not influenced by cache bandwidth because it can rely on the prediction in the return stack buffer.

The disadvantage of using a link register is that the compiler has to treat leaf functions and non-leaf functions differently, and that non-leaf functions need extra instructions for saving and restoring the leaf register on the stack.

Therefore, we will not use a link register in the ForwardCom architecture.

Separate call stack

We may have two stacks: a call stack for return addresses and a data stack for the local data of each function. A program without recursive functions will usually have a quite limited call depth so that the entire call stack, or at least the "hot" part of it, can be stored on the chip. This will improve the performance because no memory or cache operations are needed for call and return operations – at least not in the critical innermost loops of the program. It will also simplify prediction of return addresses because the on-chip rolling stack and the return stack buffer will be one and the same structure.

The call stack can be implemented as a rolling register stack on the chip. The call stack is spilled to memory if it overflows. A return instruction after such a spilling event will use the on-chip value rather than the value in memory as long as the on-chip value has not been overwritten by new calls. Therefore, the spilling event is unlikely to occur more than once in the innermost part of a program.

The pointer for the call stack should not be a general purpose register because the programmer will rarely need to access it directly. Direct manipulation of the call stack is only needed in a stack unroll event (after an exception or long jump) or a task switch.

A function does not have easy access to the return address that it was called from. Information about the caller may be supplied explicitly as a function parameter in the rare case that it is needed. There is a security advantage in hiding the return address inside the chip. This prevents overwriting return addresses in case of program errors or malicious buffer overflow attacks.

The disadvantage of having a separate call stack is that it makes memory management more complicated because there are two stacks that can potentially overflow. The size of the call stack can be predicted accurately for programs without recursive functions by using the method described on page 87.

A separate call stack may be implemented with the ForwardCom architecture. The size of the on-chip stack buffer and other details will be implementation-dependent.

Unified stack for return addresses and local data

Many current systems use the same stack for return addresses and local data. This method may be used with the ForwardCom architecture because it is simple to implement.

Conclusion for ForwardCom

A ForwardCom system may use a separate call stack or a unified stack, but not a link register. The hardware implementation of call and return instructions depends on whether there is one or two stacks. The dual stack system will be used for large processors where performance or security is important, while the unified stack system may be used in small processors where simplicity is preferred. A ForwardCom microprocessor does not have to support both systems, but the software does. The calling conventions defined on page 82 will make the software compatible with both single stack and dual stack processors. Tail calls can be implemented efficiently with a simple jump instruction regardless of the stack type.

5.3 Floating point errors and exceptions

Exceptions for floating point errors are disabled by default, but can be enabled with bits 26-29 in the numeric control register or a mask register. Enabled exceptions are caught as traps (synchronous interrupts).

It is a problem that an exception caused by a single element in a vector will interrupt the processing of the whole vector. The behavior of a program using floating point vectors will depend on the vector length in case of traps caused by a single vector element. We can rely on the generation and propagation of NAN and INF values instead of traps if we want consistent results on different processors with different vector lengths.

NAN values will be propagated through the sequence of floating point calculations. A NAN can contain a bit pattern of diagnostic information called the payload, and this bit pattern is propagated to the result. A problem arises when two different NANs are combined, for example NAN1 + NAN2. The IEEE standard (754-2008) specifies that only one of the two NAN operands is propagated to the result. This violates the fundamental principle that addition is commutative. The result can be inconsistent when a compiler swaps the two operands. Another problem with the IEEE standard is that NAN values are not propagated through the max and min instructions according to this standard.

Here, it is proposed to deviate from this unfortunate standard and output the OR combination of the input NAN payloads when multiple NAN operands are combined. This will make the propagation of NANs more useful and consistent. Different bits in the NAN payload can be used for indicating different error conditions. If multiple different error conditions have arisen in a sequence of calculations then all these conditions can be traced in the final result. This better propagation of NAN values is enabled by setting bit 22 in the numeric control register or in a mask register.

The implementation will use only one bit in the NAN payload for each error condition. A quiet NAN has bit number -1 of the significand set, while the remaining bits are available for any payload information. The ForwardCom processor puts diagnostic information in the payload if better NAN propagation is enabled by bit 22 in the numeric control register or a mask register. Bit number -2 in the significand indicates invalid arithmetic operations such as $0/0$, $0 \cdot \infty$, $\infty - \infty$, etc. Bit number -3 indicates a square root of a negative number, and other complex number results. The remaining payload bits are available for other purposes such as function libraries.

Other methods for generating error messages in function libraries are discussed on page 78.

5.4 Detecting integer overflow

There is no common standard method for detecting overflow in integer calculations. The detection of overflow in signed integer operations is a real nightmare in some programming languages like C++ (see e. g.
stackoverflow.com/questions/199333/how-to-detect-integer-overflow-in-c-c.

It would be nice to have a reliable way of detecting integer overflow and perhaps to propagate it through a series of calculations, analogous to the NAN propagation for floating point calculations, so that errors can be checked at the end of a series of calculations rather than after each operation. Compilers could support this method by offering overflow detection with a try/catch block. It is more likely that compilers will support integer overflow detection if the hardware offers a reasonable method.

The following methods have been proposed:

1. Use a few vacant bits in the mask registers for detecting and propagating overflow and other errors. This method has a number of problems that will impede out-of-order execution. The mask register will be used not only for input to each instruction but also output. Each instruction will then have two outputs rather than one. This will make the out-of-order scheduler much more complicated, and it will cause undesired dependencies when the same mask register is used for multiple instructions that otherwise would be independent.

2. Use the even-numbered elements in a vector register for normal calculation on integers and use the following odd-numbered elements for the overflow information. The overflow information is propagated together with the calculated values. This method will be efficient for scalar integer calculations, but wasteful for vectors because half the vector elements are used only for this purpose.

3. Use one element of a vector for the overflow bits of all the other elements. This method may be tempting because it does not waste as much register space as the previous method, but it will have inferior performance because of the transport delay when moving overflow bits to a distant part of a long vector.

4. Add extra bits in the vector registers for overflow information. All vector registers will have one extra overflow bit for each 32 bits of normal data. These overflow bits are preserved when a vector register is saved and restored with the save_cp and restore_cp instructions, but they are lost when the vector is saved as normal data. The behavior of the overflow bits is controlled by the following bits in the numeric control register or a mask register.
   Bit 2: detect unsigned integer overflow.
   Bit 3: detect signed integer overflow.
   Bit 4: detect floating point overflow (tentative).
   Bit 5: detect floating point invalid operations (tentative).
   Bit 6: propagate overflow information from input operands by OR'ing the result of the current instruction with the overflow bits of all vector register input operands. An extra instruction must be provided for extracting the overflow bits from a vector register.

5. Generate a trap in case of integer overflow. Use a mask register or the numeric control register as in method 4. Bit 7 enables a trap on the conditions indicated by bit 2 (unsigned integer overflow) or bit 3 (signed integer overflow). This method requires little extra code, but it is subject to the problem that the behavior of vector code depends on the vector length in case of traps, as explained in the previous chapter for floating point errors.

Method 2 is tentatively supported here with the optional instructions add_oc, etc., described on page 53.

Support for method 4 may be considered, since it would be more efficient and useful. The cost of implementing method 4 is that we will need 3% more bits in the vector registers; the save_cp and

restore_cp instructions will be more complicated; and the compiler has to check for overflow before saving vectors to memory in the normal way.

Method 5 should be supported. It is useful for integer code in general purpose registers and it is useful for verifying that overflow does not occur in vector registers.

These methods should not detect overflow in saturated arithmetic instructions and shift instructions.

5.5 Multithreading

The ForwardCom design makes it possible to implement very large vector registers to process large data sets. However, there are practical limits to how much you can speed up the performance by using larger vectors. First, the actual data structures and algorithms often limit the vector length that can be used. And second, large vectors mean longer physical distances on the semiconductor chip and longer transport delays.

Additional parallelism can be obtained by running multiple threads in each their CPU core. The design should allow multiple CPU chips or multiple CPU cores on the same physical chip.

Communication and synchronization between threads can be a performance problem. The system should have efficient means for these purposes, including speculative synchronization.

It is probably not worthwhile to allow multiple threads to share the same CPU core and level-1 cache simultaneously (this is what Intel calls hyper-threading) because this could allow a low priority thread to steal resources from a high priority thread, and it is difficult for the operating system to determine which threads might be competing for the same execution resources if they are run in the same CPU core.

5.6 Security features

Security is included in the fundamental design of both hardware and software. This includes the following features.

- A flexible and efficient memory protection mechanism.

- Optional separation of call stack and data stack so that return addresses cannot be compromised by buffer overflow.

- Each thread has its own protected memory space, except where compatibility with legacy software requires a shared memory space for all threads in an application.

- Device drivers and system functions have carefully controlled access rights. These functions do not have general access to application memory, but only to a specific block of memory that an application may share with a system function when calling it. A device driver has only access to a specific range of input/output ports and system registers as specified in the executable file header and controlled by the system core.

- A fault in a device driver should not generate a "blue screen of death", but generate an error message and close the application that called it and free its resources.

- Application programs have only access to specific resources as specified in the executable file header and controlled by the system.

- Array bounds checking is simple and efficient, using an addressing mode with built-in bounds checking or a conditional trap.

- Various optional methods for checking integer overflow.

- There is no "undefined" behavior. There is always a limited set of permissible responses to an error condition.

How to improve the security of applications and systems

Several methods for improving security are listed below. These methods may be useful in ForwardCom applications and operating systems where security is important.

Protect against buffer overflow

Input buffers must be protected against overflow. If a software-based protection is not sufficient then you may allocate an isolated block of memory for the input buffer. See page 72.

Protect arrays

Array bounds should be checked.

Protect against integer overflow

Use one of the methods for detecting integer overflow mentioned on page 63.

Protect thread memory

Each thread in an application should have its own protected memory space. See page 72.

Protect code pointers

Function pointers and other pointers to code are vulnerable to control flow hijack attacks. These include:

Return addresses. Return addresses on the stack are particularly vulnerable to buffer overflow attacks. Use a dual stack design to isolate the return stack from other data.

Jump tables. Switch/case multiway branches are often implemented as tables of jump addresses. These should use the jump table instruction with the table placed in the CONST section with read-only access. See page 25.

Virtual function tables. Programming languages with object polymorphism, such as C++, use tables of pointers to virtual functions. These should use the call table instruction with the table placed in the CONST section with read-only access. See page 25.

Procedure linkage tables. Procedure linkage tables, import tables and symbol interposition are not used in ForwardCom. See page 86.

Callback function pointers. If a function receives a pointer to a callback function as parameter, then keep this pointer in a register rather than saving it to memory.

State machines. If a state machine or similar algorithm is implemented with function pointers then place these function pointers in a constant array, use a state variable as index into this array and check the index for overflow. The compiler should have support for defining an array of relative function pointers in the CONST section and access them with the call table instruction.

Other function pointers. Most uses of function pointers can be covered by the methods described above. Other uses of function pointers should be avoided in high security applications, or the pointers should be placed in protected memory areas or with unpredictable addresses. (See Code-Pointer Integrity link).

Control access rights of application programs

The executable file header of an application program should include information about which kinds of operations the application needs permission to. This may include permission to various network activities, access to particular sensitive files, permission to write executable files and scripts, permission to install drivers, permission to spawn other processes, permission to inter-process communication, etc. The user should have a simple way of checking if these access rights are acceptable. We may implement a system for controlling the access rights of scripts as well. Web page scripts should run in a sandbox.

Control access rights of device drivers

Many operating systems are giving very extensive rights to device drivers. Rather than having a bureaucratic centralized system for approval of device drivers, we should have a more careful control of the access rights of each device driver. The system call instruction in ForwardCom gives a device driver access to only a limited area of application memory (see page 25). The executable file header of a device driver should have information about which ports and system registers the device driver has access to. The user should have a simple way of checking if these access rights are acceptable.

Standardized installation procedure

Malware protection should be an integral part of the operating system, not a third-party add on. The operating system should provide a standardized way of installing and uninstalling applications. The system should refuse to run any program, script or driver that has not been installed through this procedure. This will make it possible for the user to review the access requirements of all installed programs and to remove any malware or other unwanted software through the normal uninstallation procedure.

Chapter 6

Programmable application-specific instructions

Rather than implementing a lot of special instructions for specific applications, we may provide a means for generating user-defined instructions which can be coded in a hardware description language, e. g. VHDL or Verilog.

The microprocessor can have an optional FPGA or similar programmable hardware. This structure can be used for making application-specific instructions or functions, e. g. for coding, encryption, data compression, signal processing, text processing, etc.

If the processor has multiple CPU cores then each core may have its own FPGA. The hardware definition code is stored in its own cache for each core. The operating system should prevent, as far as possible, that the same core is used for different tasks that require different hardware codes. There may be features for allowing an application to monopolize an FPGA or part of it.

If it cannot be avoided that multiple applications use the same FPGA in the same CPU core, then the code, as well as the contents of any memory cells in the FPGA, must be saved on each task switch. This saving may be implemented as lazy, i. e. the contents is only swapped when the second task needs the FPGA structure that contains code for the first task.

There must be instructions for accessing the user-defined functions, including means for input and output, and for adapting to the latency of the user-defined functions.

Chapter 7

Microarchitecture and pipeline design

The ForwardCom instruction set is intended to facilitate a consistent and efficient design of the pipeline of a superscalar microprocessor. Instructions can have one destination operand, up to three or four source operands, a mask register, and a register specifying vector length. The last source operand can be a register, a memory operand or an immediate constant. All other operands are registers, except for memory write instructions. The total number of input registers to an instruction, including source operands, mask, base pointer, index and vector length specifier cannot be more than five.

No instruction can have more than one memory operand. No instruction can have both a memory source operand and an immediate operand, though this may be allowed in future extensions. Any extra immediate operand field can be used for option bits.

A high performance pipeline may be designed as superscalar with the following stages.

- Fetch. Fetching blocks of code from the instruction cache, one cache line at a time, or as determined by the branch prediction machinery.

- Instruction length decode. Determine the length of each instruction and identify tiny instructions. Distribute the first P instructions into each their pipeline lane, where P is the number of parallel lanes implemented in the pipeline. Excess instructions may be queued for the next clock cycle.

- Instruction decode. Identify and classify all operands, opcode and option bits. Determine input and output dependencies.

- Register allocation and renaming.

- Instruction queue.

- Put instructions into reservation station. Schedule for address calculator.

- Calculate address and length of memory operand. Check access rights.

- Read memory operand. Schedule for execution units.

- Execution units.

- Retire or branch.

It is not necessary to split instructions into micro-operations if the reading of memory operands is done in a separate pipeline stage and instructions are allowed to stay in the reservation station until the memory operand has been read.

Each stage in the pipeline should ideally require only one clock cycle. Instructions waiting for an operand should stay in the reservation station. Most instructions will use only one clock cycle in the execution unit. Multiplication and floating point addition need a pipelined execution unit with several stages. Division and square root may use a separate state machine.

Jump, branch, call and return instructions also fit into this pipeline design.

The reservation station has to consider all the input and output dependencies of each instruction. Each instruction can have up to five input dependencies and one output dependency.

There can be multiple execution units so that it is possible to run multiple instructions in the same clock cycle if their operands are independent.

An efficient out-of-order processing requires renaming of the general purpose registers and vector registers, but not necessarily the special registers.

Complex instructions and microcode should generally be avoided. We do not have an instruction for saving or restoring all registers during a task switch. Instead, the necessary instructions for saving and restoring registers are implemented as tiny instructions to reduce the size of an instruction sequence that saves all registers.

The following instructions are moderately complex: call, return, div, rem, sqrt, cmp_swap, save_cp, restore_cp. These instructions may be implemented as dedicated state machines. The same applies to traps, Interrupts and system calls.

Some current CPUs have a "stack engine" in order to predict the value of the stack pointer for a push, pop or call instruction when preceding stack operations are delayed due to operands that are not available yet. Such a system is not needed if we have a dual stack design (see page 61). Even with a single stack design, there is little need for a stack engine because push and pop operations will be rare in critical parts of the code if the function calling conventions in this document are followed (page 82).

Branch prediction is important for the performance. We may implement four different branch prediction algorithms: one for ordinary branches, one for loops, one for indirect jumps, and one for function returns. The long form of branch instructions have an option bit for indicating loop behavior. The short form of branch instructions does not have space for such a bit. The initial guess may be to assume loop behavior if the branch goes backwards and ordinary branch behavior if the branch goes forwards. This assumption may be corrected later, if necessary, by the branch prediction machinery.

The code following a branch is executed speculatively until it is determined whether the prediction was right. We may implement features for running both sides of a branch speculatively at the same time.

The ForwardCom design allows large microprocessors with very long vector registers. This requires special design considerations. The chip layout of vector processors is typically divided into "data lanes" so that the vertical transfer of data from a vector element to the corresponding vector element in another vector (i. e. same lane) is faster than the horizontal transfer of data from one vector element to another element at another position of the same vector (i. e. different lane). This means that instructions that transfer data horizontally across a vector, such as broadcast and permute instructions, may have longer latencies than other vector instructions. The scheduler needs to know the instruction latency, and this can be a problem if the latency depends on the distance of data transfer on very long vectors. This problem is addressed by indicating the vector length or the distance of data transfer for such instructions in a separate operand, which always uses the RS register field. This information may be redundant because the vector length is stored in the vector register operands, but the scheduler needs this information as early as possible. The other register operands are typically not ready until the clock cycle where they go to the execution unit, while the vector length is typically known earlier. The microprocessor can read the RS register at the address calculation stage in the pipeline, where it also reads any pointer, index register and vector length for memory operands. This allows the scheduler to predict the latency a few clock cycles in advance. The instruction set provides the extra information about vector length or data transfer length in RS for all instructions that involve horizontal data transfer, including memory broadcast, permute, insert, extract and shift instructions, but not broadcasting of immediate constants.

The data path to the data cache and memory should be quite wide, possibly matching the maximum

vector length, because cache access and memory access are typical bottlenecks.

Chapter 8

Memory model

The address space is using unsigned 64-bit addresses and 64-bit pointers. Future extension to 128-bit addresses is possible, but this will probably not be relevant in a foreseeable future.

Absolute addresses are rarely used. Most data objects, functions and jump targets are addressed with signed offsets of 32 bits or less relative to some reference point contained in a 64-bit pointer. This pointer can be the instruction pointer (IP), the data section pointer (DATAP), the stack pointer (SP), or a general purpose register.

An application can have access to the following sections of data:

- Program code (CODE). This memory block is executable with or without read access, but without write access. The CODE section can be shared between multiple processes running the same program.

- Constant program data (CONST). This contains constants and tables used by the program without write access. It may be shared between multiple processes.

- Static read/write program data sections, which can be initialized (DATA) and uninitialized (BSS). This is used for global data and for static data inside functions. Multiple instances are needed if multiple processes are running the same code.

- Stack data (STACK). This is used for non-static data inside functions. Each process or thread has its own stack, addressed relative to the stack pointer. The stack grows downward from high to low addresses when data are added to the stack.

- Program heap (HEAP). Used for dynamic memory allocation by an application program.

- Thread data (THREADD). Allocated when a thread is created and used for thread-local static data and thread environment block.

References within the CODE section use 8-bit, 16-bit, 24-bit and 32-bit signed references relative to the instruction pointer, scaled by the code word size which is 4 bytes.

The CONST section is preferably placed immediately before the CODE section. Data in the CONST section are mostly addressed relative to the instruction pointer with no scale factor. (In case of a pure Harvard architecture, the CONST section may be placed in readable program memory to be addressed relative to the instruction pointer, or it may be placed in data memory and addressed relative to DATAP).

The DATA and BSS sections are addressed relative to the data section pointer (DATAP) which is a special register that points to some reference point in these sections. The preferred reference point is where DATA ends and BSS begins. Multiple running instances of the same program will have different values of the data section pointer. The CODE and CONST sections contain no direct references to DATA or BSS, only references relative to the data section pointer. This makes it possible for multiple processes to share the same CODE and CONST sections, but have each their private DATA and

BSS sections without the need for virtual address translation. The DATA and BSS sections can be placed anywhere in the address space independently of where CONST and CODE are placed.

STACK data are addressed relative to the stack pointer (SP). Heap data are addressed through pointers provided by the heap allocation function.

Thread data are addressed relative to a register called thread environment block pointer (THREADP), which is separate for each thread in the process. The thread environment block may be allocated on the stack when a new thread is created.

The STACK, DATA, BSS, HEAP and THREADD data sections are preferably kept together in one contiguous block in order to optimize caching and memory management.

This model allows the program to access up to 8 GB of CODE, 2 GB of CONST, 2 GB of DATA, 2 GB of BSS, 2 GB of THREADD, an almost unlimited size of STACK with 2 GB frames, and an almost unlimited amount of HEAP data. A pointer to the CONST section is provided in the thread environment block in order to access CONST data in the rare case that the distance between code and data exceeds 2 GB or in order to avoid address relocation.

The end of the combined data memory block must have an unused space of the same size as the maximum vector length. This will enable the restore_cp instruction to read more than necessary when restoring a vector of unknown length. It will also allow a function that searches for the end of a zero-terminated string to read one vector-length piece of the string at a time without causing access violation by reading into unavailable memory space.

Most microprocessor systems have the stack growing backward. The ForwardCom system has the same, but mainly for a different reason. When a vector register is saved on the stack, it is stored as the length followed by the amount of data indicated by the length. When the vector register is restored (using the restore_cp instruction), it is necessary to read the length followed by the data. The stack pointer must point to the low end where the length is stored, otherwise it would be impossible to find where the length is stored.

8.1 Thread memory protection

Each thread must have its own stack. The thread data (THREADD) may be placed on this stack. The ForwardCom system allows inter-thread memory protection. The stack data of the main thread of a program is accessible to all its child threads, but all other threads in the program can have private data which is not accessible to any other threads, not even to the main thread. Any communication and synchronization between threads must use static memory or memory belonging to the main thread.

It is recommended to use this inter-thread memory protection in all cases except where legacy software requires one memory space shared by all threads.

Isolated memory blocks

It is possible to make a system function that allocates an isolated memory block surrounded by inaccessible memory on both sides. Such a memory block, which will be accessible only to a specific thread, can be used for example for an input buffer in cases where security requirements are high. Each thread can have only a limited number of such protected memory blocks because of the limited size of the memory map.

8.2 Memory management

It is a design goal to minimize memory fragmentation and to minimize the need for virtual address translation. Current designs often have very complicated memory management systems with multi-level address translation, large translation-lookaside-buffers (TLB), and huge page tables. We want to

replace the TLB, which has a large number of fixed-size memory blocks, by a memory map with a few memory blocks of variable size. In most cases, the main thread of an application will only need three blocks of memory: CONST (read only), CODE (execute only), and the combined STACK+DATA+BSS+HEAP (read-write). A child thread needs one more entry for its private stack. Similar blocks are defined for system code.

A memory map with such a limited number of entries can easily be implemented on the chip in a very efficient way and it can easily be changed on task switches. Each process and each thread must have its own memory map. The memory is not organized into fixed-size pages.

The memory map supports virtual address translation in the form of a constant offset that defines the distance between the virtual address and the physical address for each map entry. The hardware should not waste time and power on virtual address translation when it is not used.

A limited number of extra entries are provided in the memory map to deal with cases where the memory becomes fragmented, but memory fragmentation can be avoided in most cases. The following techniques are provided to simplify memory management and avoid memory fragmentation:

- There is only one type of function libraries which can be used for both static and dynamic linking. These are linked with a mechanism that keeps the CONST, CODE and DATA sections contiguous with the similar sections of the main program in most cases. This technique is described on page 86 below.

- The required stack size is calculated by the compiler and the linker so that stack overflow can be avoided in most cases. This technique is described on page 87.

- The operating system can keep statistical records of the heap use of each program in order to predict the required heap size. The same technique can be used for predicting stack use in cases where the required stack size cannot be predicted exactly (e. g. recursive function calls).

The memory space may become fragmented despite the use of these techniques. Problems that can result in memory fragmentation are listed below.

- Recursive functions can use unlimited stack space. We may require that the programmer specifies a maximum recursion level in a pragma.

- Allocation of variable-size arrays on the stack using the alloca function in C. We may require that the programmer specifies a maximum size.

- Runtime linking. The program can reserve space for loading and linking function libraries at run time (see page 86). The memory may become fragmented if the memory space reserved for this purpose turns out to be insufficient.

- Script languages and byte code languages. It is difficult to predict the required size of stack and heap when running interpreted or emulated code. It is recommended to use a just-in-time compiler instead. Self-modifying scripts cannot be compiled. The same problem can occur with large user-defined macros.

- Unpredictable number of threads without protection. The required stack size for a thread may be computed in advance, but in some cases it may be difficult to predict the number of threads that a program will generate. Multiple threads will mostly share the same code sections, but they need separate stacks. The stack of a thread can be placed anywhere in memory without problems if inter-thread memory protection is used. But if memory is shared between threads and the number of threads is unpredictable then the shared memory space may become fragmented.

- Unpredictable heap size. Programs that process large amounts of data, e. g. multimedia processing, may need a large heap. A heap can use discontiguous memory, but this will require extra entries in the memory map.

- Lazy loading and code overlay. A large program may have certain code units that are rarely used and loaded only when needed. Lazy loading can be useful to save memory, but it may require virtual memory translation and it may cause memory fragmentation. A straightforward solution is to implement such code units as separate executable programs.

- Hot patching, i. e. updating of code while it is running.

- Shared memory for inter-process communication. This requires extra entries in the memory map as explained below.

- Many programs running. The memory can become fragmented when many programs of different sizes are loaded and unloaded randomly or swapped to memory.

A possible remedy against overflow of stack and heap is to place the STACK, DATA, BSS and HEAP data together (in this order) in an address range with large unused virtual address spaces below and above, so that the stack can grow downwards and the heap can grow upwards into the vacant spaces. This method can avoid fragmentation of the virtual address space, but not the physical address space. Fragmentation of the physical address space can be remedied by moving data from a memory block of insufficient size to another block that is larger. This method has the cost of a time delay when the data are moved.

If runtime linking runs into memory problems and lack of memory map entries then it is allowed to mix CONST and CODE sections together in a common section with both read and execute access. If a library function contains constant data that originate from an untrusted source, while the code is trusted, then it is preferred to put the untrusted data into the DATA section rather than the CONST section in order to prevent execution of malicious code placed in the CONST section.

Shared memory can be used when there is a need to transfer large amounts of data between two processes. One process shares a part of its memory with another process. The receiving process needs an extra entry in its memory map to indicate read and/or write access rights to the shared memory block. The process that owns the shared memory block does not need any extra entry in its memory map. There is a limit to how many shared memory blocks an application can receive access to, because we want to keep the memory map small. If one program needs to communicate with a large number of other programs then we can use one of these solutions: (1) let the program that needs many connections own the shared memory and give each of its clients access to one part of it, (2) run multiple threads in (or multiple instances of) the program that needs many connections so that each thread has access to only one shared memory block, (3) let multiple communication channels use the same shared memory block or parts of it, (4) communicate through function calls, (5) communicate through network sockets, or (6) communicate through files.

Executable memory cannot be shared between different applications. The mechanism of interprocess calls must be used if one application needs to call a function in another application. This is described on page 78.

We can probably keep memory fragmentation so low, by using the principles discussed here, that a relatively small memory map for each thread will be sufficient to cover normal cases. This will be much more efficient than the large TLB and multilevel address translation of current designs. It will save silicon space and power, and we can avoid the cost of TLB misses and page faults, and it will make task switches very fast.

Chapter 9

System programming

The system instructions have not been fully defined yet. There is more work to do making an efficient system design. However, the first experimental implementations of ForwardCom will be without operating system so the system design does not have to be fixed yet. It is preferred to spend more time on optimizing the system design rather than to define a complete standard at this early stage of development.

There should be at least three different levels of privilege:

- The system core has the highest privilege level. Memory management and thread scheduling takes place here. This is the only part that can modify memory maps and control access rights at the lower levels.

- Device drivers and system plugin modules have carefully controlled access rights. A structure similar to the memory map (see page 72) gives a device driver access to the particular range of input/output ports and system registers that it needs. A user application can give a device driver read and write access to a specific range of the data memory it owns. This is done through the system call instruction. A device driver has no access to the code memory of the application that calls it. This means that callback function pointers cannot be used with system calls.

- An application program has access to only the memory that is allocated to it or shared with it. Memory belonging to a thread is usually not shared with other threads in the same process. Application programs have access to a few system registers and no input/output ports.

Transitions between these levels are managed by the system call and system return instructions and by traps and interrupts.

There are various system registers for control purposes. In addition, there are two sets of registers used for temporary storage, one set for the device driver level and one for the system core level. The temporary registers for the device driver level are cleared for security reasons every time a device driver is called. These registers are used mainly for temporary saving of the general purpose registers.

9.1 Memory map

There are three kinds of memory access: read, write and execute access. These kinds of access are separate, but can be combined. For example, execute access does not imply read access. Write access and execute access should not normally be combined, because self-modifying code is discouraged.

The memory map is stored in the CPU chip. Each entry has three fields: A virtual address (up to 64 bits), access rights (3 bits), and an addend for address translation (up to 64 bits). There is no memory paging. Instead, the memory blocks have variable sizes.

The entries in the memory map must be kept sorted at all times so that each memory block ends where the next block begins. The addresses must be divisible by 8. Each thread has its own memory map. A typical memory map for an application thread may look like this.

Table 9.1: Example of memory map

Start address	Access	Addend	Comment
0x10000	Read	0	CONST section
0x10100	Execute	0	CODE section
0x10800	None	0	Belongs to other processes
0x20000	Read, Write	0	Main STACK, DATA, BSS, and HEAP sections
0x24000	None	0	Belongs to other processes
0x30000	Read, Write	0	Thread STACK, thread environment block, and tread static data
0x32000	None	0	The rest belongs to other processes

There may be a few further entries for memory blocks shared between processes and for secure isolated memory blocks. A virtual memory block may have multiple entries in case the memory becomes fragmented. The addends are used for keeping the virtual addresses of the block contiguous while the physical addresses are noncontiguous. The start addresses are virtual memory addresses.

The size of the memory map is variable. The maximum size is implementation dependent. There are at least three memory maps on the chip, one for each privilege level. This makes transitions between the levels fast. The chip space used for memory maps may be reconfigurable so that the memory maps of multiple processes can remain on the chip in case the memory maps are small. This makes task switching faster.

The memory maps are controlled at the system core level. The instructions read_memory_map and write_memory_map use the vector loop mechanism for fast manipulation of memory maps.

The methods described on page 72 for avoiding memory fragmentation are important for keeping the memory maps small.

Task switches will be very fast because we have replaced the large page tables and translation-lookaside-buffer (TLB) of traditional systems with a small on-chip memory map. This makes the system suitable for real-time operating systems.

9.2   Call stack

It is possible to have either a unified stack for function data and return addresses or two separate stacks. See page 60. ForwardCom currently supports both systems. The two-stack system is safer and more efficient, while the single-stack system may be used for small processors where the simpler single-stack system is preferred.

The two-stack system has the call stack stored inside the CPU rather than in RAM memory. A method is required for saving this stack to memory when it is full. This method may be similar to the method used for saving the memory map, as described above, using vector-size memory access. It should be possible to manipulate the call stack for task switches and for stack unrolling in the exception handler.

9.3   System calls and system functions

Calls to system functions are made with a system call instruction (sys_call). The system call instruction does not use addresses, but ID numbers. Each ID number consists of a function ID in the lower half and a module ID in the upper half. The module ID identifies a system module or device driver. The system core has ID = 0. Each part of the ID can be either 16 bits or 32 bits so that the combined ID is either 32 bits or 64 bits.

System add-on modules and device drivers do not necessarily have fixed ID numbers because this would require some central authority to assign these ID numbers. Instead, the program will have to ask for the ID number by giving the name of the module. The functions within a module can have fixed or variable ID numbers.

There will be a system function (with a fixed ID number) which takes the names of module and function as input and returns the ID number. The ID number can be retrieved in this way before the first call to the function.

The ID number of a system function can be put into the program in three ways:

1. The most important system functions have fixed ID numbers which can be inserted at compile time.

2. The ID number can be found at load time in the same way as load-time linking works. This is described on page 86. The loader will find the ID number and insert it in the code before running the program.

3. The ID number is found at run time before the first call to the desired function.

The calling convention for system functions is the same as for other functions, using registers for parameters and for return value. The registers used for parameters are determined by the general calling convention. The calling conventions are described on page 82. The parameter registers should not be confused with the operands for the system call instruction.

The system call instruction has three operands. The first operand is the combined ID, contained in a register (RT) or an immediate constant. The second operand (RD) is a pointer to a memory block that may be used for transferring data between the calling program and the system function. The third parameter (RS) is the size of this memory block. The last two parameters must be divisible by 8.

The calling thread must have access rights to the memory block that it shares with the system function. This can be read access or write access or both. These access rights are transferred to the system function. The system function has no access rights to any other part of the application's memory.

It is not possible to use callback function pointers with a system call because executable memory cannot be shared with a system function. Instead, the system function can call an exported function provided by the application, using the method for inter-process calls, described below.

Device driver functions should preferably have separate stacks. The system call goes first to the system core which assigns a stack to the device driver function and makes a memory map for it before dispatching the call to the desired function. Preferably, no stack is used during this dispatching. The two registers identifying a shared memory block are copied to special registers which are accessible to the called function. The system function runs in the same thread as the application that called it, but not with the same stack.

The old values of instruction pointer, stack pointer, DATAP and memory map are saved in system registers, to be restored by the system return instruction.

System functions, device drivers and interrupt handlers are allowed to use all general purpose registers and vector registers if they are saved and restored according to the normal calling conventions. Interrupt handlers must save and restore all registers they use.

A method is provided to get information about the register use of system functions so that it is possible to call them using the register usage conventions of either method 1 or method 2, described on page 84. The stack use of system functions is irrelevant for the caller because they do not use the stack of the calling application program.

Some important system functions must be standardized and must be available in all operating systems. This will make it possible, for example, to make a third-party function library that works in

all operating systems, even if this library needs to call system functions. It will also make it easier to adapt a program for different operating systems. The list of system functions that might be standardized includes functions for thread creation, thread synchronization, setting thread priority, memory allocation, time measurement, system information, access to environment variables, etc.

There should be a selection of system libraries providing the most common user interface forms, such as graphical user interface, console mode, and server mode. These user interface system libraries should be provided for each operating system that the architecture can run on, so that the same executable program can run in different operating systems simply by linking with the appropriate user interface library at load time. Such user interface libraries may be based on existing platform-independent GUI libraries such as, e. g., wxWidgets or QT. All user interface libraries must support the error_message function mentioned below.

9.4   Inter-process calls

Inter-process calls are mediated by a system function. This works in the following way. An application program can export a function with an entry in its executable file header. Another application can get access to this exported function by calling a system function that checks for permission and switches the memory map, the DATAP and THREADP registers and the stack pointer before calling the exported function, and switches back before returning to the caller. The call will appear as a separate thread to the called program. The general purpose registers and vector registers can be used for parameters and return value in the same was as for normal functions. This mechanism does not generate any shared memory between caller and callee. Therefore, the exported function must use only simple types that fit into registers for its parameters and return type. A block of memory can be shared between the two processes as described on page 74.

9.5   Error message handling

There is a need for a standardized way of reporting errors that occur in a program. Many current systems fail to satisfy this need, or they use methods that are not portable or thread-safe. In particular, the following situations would benefit from such a standard.

1. A function library detects an error, for example an invalid parameter, and needs to report the error to the calling program. The calling program will decide whether to recover from the error or terminate.

2. A trap is generated because of a numerical error. The program fails to catch it as an exception, or the programming language has no support for structured exception handling. The operating system must make an informative error message.

3. A program can run in different environments that require different forms of error handling.

4. A function library in source code form, a class library, or any other piece of code needs to report an error without knowing which user interface paradigm is used (e. g. console mode or graphical user interface). It needs a standardized way of reporting the error to the operating system or to the user interface framework, which must present an error message to the user in the way that is appropriate for the user interface (e. g. pop up a message box, print to stderr, print to a log file, or send a message to an administrator).

It is proposed to define a standard library function named error_message for this purpose. All user interface frameworks must define this function. It is possible to automatically choose between different versions of this function at run time depending on system settings, using the function dispatch feature described on page 87. The main program may override this function by defining its own function with the same name.

The error_message function must have the following parameters: a numerical error code, a string pointer giving an error message, and another string pointer giving the name of the function where the error occurred. These strings are coded as zero-terminated UTF-8 strings. The error message is in the English language by default. It is not reasonable to require support for many different languages (see this link for a discussion of problems with internationalization). Instead, a manual in the desired language can contain a list of error codes.

The error message string may include numerical values and diagnostic information, such as the value of a parameter that is out of range.

The error_message function may or may not return. If it returns then the function that called it must return in a graceful way. The error_message function may alternatively terminate the application or it may raise an exception or trap which is handled by the operating system in case the exception is not caught by the program.

Chapter 10

Standardization of ABI and software ecosystem

The goal of ForwardCom is a vertical redesign that defines new standards not only for the instruction set, but also for the software that uses it. This will have the following advantages.

- Different compilers will be compatible. The same function libraries can be used with different compilers.

- Different programming languages will be compatible. It will be possible to compile different parts of a program in different programming languages. It will be possible to compile a function library in a programming language different from the program that uses it.

- Debuggers, profilers and other development tools will be compatible.

- Different operating systems will be compatible. It will be possible to use the same function libraries in different operating systems, except if they use system-specific functions.

The previous chapter described standardization of system calls, system functions, and error messaging. The present chapter discusses standardization of the following aspects of the software ecosystem.

- Compiler support.

- Binary data representation.

- Function calling conventions.

- Register usage conventions.

- Name mangling for function overloading

- Binary format for object files and executable files.

- Format and link methods for function libraries.

- Exception handling and stack unrolling.

- Debug information.

- Assembly language syntax.

10.1 Compiler support

Compilers can have three different levels of support for variable-length vector registers.

Level 1

The compiler will not use variable-length vectors. The compiler can call a vector function in a function library with a scalar parameter if the function is not available in a scalar version.

Level 2

The compiler can call vector functions, but not generate such functions. The compiler can vectorize a loop automatically and call a vector library function from such a loop.

Level 3

Full support. The compiler supports data types for variable-length vectors. These data types can be used for variables, function parameters and function returns. Variable-length vectors can not be included in structures, classes or unions because such composite types must have known sizes. Support for variable-length vectors in static and global variables is optional. General operations on variable-

length vectors can be specified explicitly, including options for applying boolean vector masks.

Other compiler features

The compiler may support pointer arithmetic on function pointers in order to write compact call tables with relative addresses explicitly. The difference between two function pointers should be scaled by the code word size, which is 4. Without this feature, the function pointers have to be type cast to integer pointers and back again.

The compiler may have support for detecting integer and floating point overflow and other numerical errors in try-catch blocks using one of the methods discussed on page 63.

The compiler may support array bounds checking, using the indexed addressing mode with bounds or the conditional trap instruction.

10.2 Binary data representation

Data are stored in little-endian form in RAM memory. See page 60 for the rationale.

Integer variables are represented with 8, 16, 32, 64, and optionally 128 bits, signed and unsigned. Signed integers use 2's complement representation. Integer overflow wraps around, except in saturated arithmetic instructions.

Floating point numbers are coded with single (32-bit), double (64-bit) and optionally quadruple (128-bit) precision, following the IEEE Standard 754-2008 or any later standard. Half precision (16-bit) is optionally used in immediate constants. Calculation on half precision is not supported, but conversion between half and single precision is optionally supported.

Floating point NAN variables can contain diagnostic information about the cause of errors as discussed on page 62.

Boolean variables are stored as integers of at least 8 bits with the values 0 and 1 for FALSE and TRUE. Only bit 0 of the boolean variable is used, while the other bits are ignored. This rule makes it possible to use boolean variables as masks and to implement boolean functions such as AND, OR, XOR, and NOT in an efficient way with simple bitwise instructions, rather than the method used in many current systems that have a branch for each variable to check if the whole integer is nonzero. A branch instruction is needed in the compilation of expressions like (A && B) and (A || B) only if the evaluation of B has side effects.

All variables not bigger than 8 bytes should be kept at their natural alignment.

Arrays not smaller than 8 bytes must be aligned to addresses divisible by 8. It may be recommended to align large arrays by the cache line size.

Multidimensional arrays are stored in row-major order, except where the programming language makes this impossible.

Text strings may be stored in language-dependent forms, but a standardized form is needed for system functions and for functions that are intended to be compatible with all programming languages. The proposed standard uses UTF-8 encoding. The length of the string may be determined by a terminating zero or a length specifier, or both. The rationale is this. The CPU processing time is insignificant for text strings of a length suitable for human reading. The priority is therefore on compactness. Compactness matters if the string is stored in a file or transmitted over a network. UTF-8 is more compact than UTF-16 in most cases, though less compact for some Asian languages. UTF-8 is the most common encoding used on the Internet.

10.3 Further conventions for object-oriented languages

Object oriented languages require further standards for the binary representation of special features such as virtual function tables, runtime type identification, member pointers, etc.

These details must be standardized within each programming language for the sake of compatibility between different compilers, and if possible also between different programming languages that have compatible features.

Member pointers should be implemented in a way that prioritizes good performance in the general case where only a simple offset (to data) or a pointer (to a function) is required, while additional information for contrived cases of multiple inheritance is added only when needed.

10.4 Function calling convention

Function calls will use registers for parameters as much as possible. Integers of up to 64 bits, pointers, references, and boolean scalars are transferred in general purpose registers. Vector parameters can have variable length. Floating point scalars, vectors of any type with a fixed length of up to 16 bytes, and vectors of variable length are transferred in vector registers.

The first 16 parameters to a function that fit into a general purpose register are transferred in register r0 – r15. The first 16 parameters that fit into a vector register are transferred in v0 – v15. The length of a variable-length vector parameter is contained in the same vector register that contains the data.

Composite types are transferred in vector registers if they can be considered "simple tuples" no bigger than 16 bytes. A simple tuple is a structure or class or encapsulated array for which all non-static elements have the same type, which is not a pointer. A union is treated as a structure according to its first element.

Parameters that do not fit into a single register are transferred by a pointer to a memory object allocated by the caller. This applies to: structures and classes with elements of different types, or bigger than 16 bytes. It also applies to objects that require special handling such as a non-standard copy constructor or destructor, and objects that require extra implicit storage such as tables of virtual member functions. It is the responsibility of the caller to call any copy constructor and destructor.

If there are not enough registers for all parameters then the additional parameters are provided in a list, which can be stored anywhere in memory. A pointer to this parameter list is transferred in a general purpose register. Such a list is also used if there is a variable argument list. There can be no more than one parameter list, as the same list is used for all purposes.

The rules for a parameter list are as follows. A parameter list is used if there are more than 16 parameters that fit into a general purpose register, if there are more than 16 parameters that fit into a vector register, or if there is a variable argument list. If there are less than 16 general purpose parameters then these parameters are put in general purpose registers, and the next vacant general purpose

register is used as pointer to the list. If there are 16 or more general purpose parameters, and a parameter list is needed for any reason, then the first 15 general purpose parameters are put in r0-r14, the list pointer is in r15, and the remaining general purpose parameters are put in the list. If there are more than 16 vector parameters then the first 16 vector parameters are put in v0-v15 and the remaining vector parameters are put in the list. All parameters in the list are placed in the order that they appear in the function definition, regardless of type. Variable arguments are placed last in the list because they always appear last in a function definition.

The list consists of entries of 8 bytes each. A general purpose parameter uses one entry. A vector parameter with a constant size of 8 bytes or less uses one entry. A vector parameter with a constant size of more than 8 bytes or a variable size uses two entries in the list. The first entry is the length (in bytes) and the second entry is a pointer to an array containing the vector. A parameter that would not fit into a register, if one was vacant, is transferred by a pointer in the list according to the same rules as if the pointer was in a register.

The parameter list belongs to the called function in the sense that it is allowed to modify parameters in the list if they are not declared as constant parameters. The same applies to arrays and objects with a pointer in the list. The caller can rely on parameters in the list being unchanged only if they are declared constant. The caller must put the list in a place where it cannot be modified by other threads.

The function return value is in r0 or v0, using the same rules as for function parameters. Multiple return values (if allowed by the programming language) are treated as tuples if possible and returned in v0. Multiple return values of different types may be returned in multiple registers, but it is generally preferred to treat multiple return values as a structure for the sake of compatibility with other programming languages that do not allow multiple return values.

A return value that does not fit into a register is returned in a space allocated by the caller through a pointer transferred by the caller in r0 and returned in r0. Any constructor is called by the callee.

A "this" pointer for a class member function is transferred in r0, except if r0 is used for a return object, where the "this" pointer is transferred in r1.

Rationale

It is much more efficient to transfer parameters in registers than on the stack. The present proposal allows up to 32 parameters, including variable length vectors, to be transferred in registers, leaving 15 general purpose registers and 16 vector registers for the function to use for other purposes while handing the parameters. This will cover almost all practical cases, so that parameters only rarely need to be stored in memory.

Nevertheless, we must have precise rules for covering an unlimited number of parameters if the programming language has no limit to the number of parameters. We are putting any extra parameters in a list rather than on the stack as most other systems do. The main reason for this is to make the software independent of whether there is a separate call stack or the same stack is used for return addresses and local variables. The addresses of parameters on the stack would depend on whether there is a return address on the same stack. The list method has further advantages. There will be no disagreement over the order of parameters on the stack and whether the stack should be cleaned up by the caller or the callee. The list can be reused by the caller for multiple calls if the parameters are constant, and the called function can reuse a variable argument list by forwarding it to another function. The function is guaranteed to return properly without messing up the stack even if caller and callee disagree on the number of parameters. Tail calls are possible in all cases regardless of the number and types of parameters.

10.5 Register usage convention

Most systems have rules that certain registers have callee-save status. This means that a function must save these registers and restore them before it returns, if they are used. The caller can then rely on these registers being unchanged after the function call.

Current systems have a problem with assigning callee-save status to vector registers. Future CPU versions may make the vector registers longer, and the instructions for saving the longer registers have not been defined yet. Some systems now have callee-save status on part of a vector register because of poor foresight. It is impossible in current systems to save a vector register in a way that will be compatible with future extensions.

This problem is solved by the ForwardCom design with variable vector length. It is possible to save and restore a vector register of any length, even if this length was not supported at the time the code was compiled. It is also possible to know how much of a long vector register is actually used, because the length of a vector is saved in the register itself, so that we only need to save the part of the register that is actually used. The save_cp and restore_cp instructions are designed for this purpose (see page 46). Unused vector registers will use only little space for saving.

It still takes a lot of cache space to save the vector registers if they are long. Therefore, we want to minimize the need for saving registers. It is proposed to have two different methods to choose between. These methods are explained here.

Method 1

This is the default method which can be used in all cases, but not the most efficient method.

The rule is simply that registers r16 − r31 and v16 − v31 have callee-save status.

A function can use registers r0 − r15 and v0 − v15 freely. Sixteen registers of each type will be sufficient for most functions. If the function needs additional registers, it must save them.

All system registers and special registers have callee-save status, except in functions that are intended for manipulating these registers.

Method 2

It will be more efficient if we actually know which registers are used by each function. If function A calls function B, and A knows which registers are used by B, then A can simply choose some registers that are not used by B for any data that it needs to save across the call to B. Even a long chain of nested function calls can avoid the need to save any registers as long as there are enough registers.

If function A and B are compiled together in the same process then the compiler can easily manage this information. But if A and B are compiled separately, then we need to store the necessary information about which registers are used. This is possible with the object file format described on page 85. The information about register use must be saved in the compiled object file or library file, not in some other file that could possibly come out of sync.

Function B is preferably compiled first into an object file. This object file must contain information about which registers are modified by function B. The necessary information is simply a 64-bit number with one bit for each register that is modified (bit 0-31 for r0-r31, and bit 32-63 for v0-v31). Any registers used for parameters and return value are also marked if they are modified by the function.

When function A is compiled next, the compiler will look in the object file for B to see which registers it modifies. The compiler will choose some registers not modified by B for data that need to be saved across the call to B. Registers that are modified by B can advantageously be used in A for temporary variables that do not need to be saved across the call to B. Likewise, it will be advantageous to use the same register for multiple temporary variables if their live ranges do not overlap, in order to modify as few registers as possible. The object file for A will contain a list of registers modified by A, including all registers modified by B and by any other function that A may call. The object file for A

contains a reference to function B. This reference must contain information about which registers A expects B to modify. If B is later recompiled, and the new version of B modifies more registers, then the linker will detect the discrepancy and prompt for a recompilation of A.

If, for some reason, A is compiled before B or no information is available about B when A is compiled, then the compiler will have to make assumptions about the register use of B. The default assumption is as specified in method 1. Function A may later be recompiled if B violates these assumptions, or simply to improve efficiency.

If two functions A and B are mutually calling each other then the easiest solution is to rely on method 1. The functions should still include the information about register use in their object files.

The compiler should preferentially allocate the lower registers first in order to minimize the problem that different library functions use different registers. It may optionally skip r7 and v7 for the caller to use for masks.

The main program function is allowed to use method 2 and to modify all registers if it includes the necessary information in its object file.

Object files that are contained in a function library must include the information about register use.

System functions and device drivers cannot be accessed in the same way as normal library functions (see page 76). System functions must obey the rules for method 2, but the system should provide a method for getting information about the register use of each system function. This can be useful for just-in-time compilers.

10.6   Name mangling for function overloading

Programming languages that support function overloading use internal names with prefixes and suffixes on the function names in order to distinguish between functions with the same name but different parameters or different classes or namespaces. Many different name mangling schemes are in use, and some are undocumented. It is necessary to standardize the name mangling scheme in order to make it possible to mix different compilers or different programming languages.

The most common name mangling schemes are Microsoft and Gnu. The Microsoft scheme uses characters that cannot occur in function names (?@$). This prevents name clashes, but makes it impossible to call the mangled name directly or to translate e. g. C++ to C. The Gnu scheme generates mangled names that look unwieldy, but contain no special characters that prevent calling the mangled name directly. Therefore, the proposal is to use the Gnu mangling scheme (version 4 or later) with necessary additions for variable-length vectors, etc.

Functions with mangled names may optionally supplement the mangled name with the simple (non-mangled) name as a weak public alias in the object file. This makes it easier to call the function from other programming languages without name mangling. The weak linking of the alias prevents the linker from making error messages for duplicate names, unless a call to the name is ambiguous.

10.7   Binary format for object files and executable files

The executable file format must be standardized. The most flexible and well-structured format in common use is probably ELF. It is proposed to use ELF format for object files, function libraries, and executable files.

The details of an ELF format for ForwardCom are specified in a file named elf_forwardcom.h. This specification includes details for section types, symbol types, relocation types, etc. Additional information about register use (see page 84) and stack use (see page 87) is added to the file format.

File names must have extensions that indicate their type. It is proposed to use the following extensions. Assembly code: .as, object file: .ob, library file: .li, executable file: .ex.

10.8 Function libraries and link methods

Dynamic link libraries (DLLs) and shared objects (SOs) are not used in the ForwardCom system. Instead, we will use only one type of function libraries that can be used in three different ways:

1. Static linking. The linker finds the required functions in the library and copies them into the executable file. Only the parts of the library that are actually needed by the specific main program are included. This is the normal way that static libraries are used in current systems (.lib files in Windows, .a files in Unix-like systems such as Linux, BSD, and Mac OS).

2. Load-time linking. The library may be distributed separately from the executable file. The required parts of the library are loaded into memory together with the executable file, and all links between the main executable and the library functions are resolved by the loader in the same way as for static linking.

3. Run-time linking. The running program calls a system function that returns a pointer to the library function. The required function is extracted from the library and loaded into memory, preferably at a memory space reserved for this purpose by the main program. Any reference from the newly loaded function to other functions, whether already loaded or not, can be resolved in the same way as for static linking.

These methods will improve the performance and remedy many of the problems that we encounter with the traditional DLLs and SOs. A typical program in Windows and Unix systems will require several DLLs or SOs when it is loaded. These dynamic libraries will all be loaded into each their memory block, using an integral number or memory pages each, and possibly scattered over the memory space. This leads to a waste of memory space and poor caching. A further performance disadvantage with shared objects is that they use procedure linkage tables (PLT) and global offset tables (GOT) for all accesses to functions and variables in order to support the rarely used feature of symbol interposition. This requires a lookup in the PLT or GOT for every access to a function or variable in the library, including internal references to globally visible symbols.

The ForwardCom system replaces the traditional dynamic linking with method 2 above, which will make the code just as efficient as with static linking because the library sections are contiguous with the main program sections, and all access is immediate with no intermediate tables. The time required to load the library will be similar to the time required for dynamic linking because the bottleneck will be disk access, not calculation of function addresses.

A DLL or SO can share its code section (but not its data section) between multiple running programs that use the same library. A ForwardCom library can share its code section between multiple running instances of the same program, but not between different programs. The amount of memory that is wasted by possibly loading multiple instances of the same library code is more than compensated for by the fact that we are loading only the part of the library that is actually needed and that the library does not require its own memory pages. It is not uncommon in Windows and Unix systems to load a dynamic library of one megabyte and use only one kilobyte of it.

The load-time linking (method 2 above) is efficient in the ForwardCom system because of the way relative addresses are used. The main program typically contains a CONST section immediately followed by a CODE section. The CONST section is addressed relative to the instruction pointer so that these two sections can be placed anywhere in memory as long as they have the same position relative to each other. Now, we can place the CONST section of the library function before the CONST section of the main program, and the CODE section of the library function after the CODE section of the main program. We don't have to change any cross-references in the main program. Only cross references between the main program and the library function and between the CODE and CONST sections of the library function have to be calculated by the loader and inserted in the code.

A library function does not necessarily have any DATA and BSS sections. In fact, a thread-safe function has little use of static data. However, if the library function has any DATA and BSS sections,

then these sections can be placed anywhere within the $\pm$ 2GB range of the DATAP pointer. The references in the library function to its static data have to be calculated relative to the point that DATAP points to; but no references to data in the main program have to be modified when a library is added as long as DATAP still points to the border between the DATA and BSS sections of the main program.

The combined main program and library file can now be loaded into any vacant spaces in memory. It will need only three entries in the memory map: (1) the combined CONST sections of library and main program, (2) the combined CODE sections of main program and library functions, and (3) the combined STACK, DATA, BSS, and HEAP of the main program and the library functions.

Run-time linking works slightly differently. The reference from the main program to the library function goes through a function pointer that is provided when the library is loaded. Any references the other way – from the library function to functions or global data in the main program – can be resolved in the same way as for method 1 and 2 or through pointer parameters to the function. The main program should preferably reserve space for the CONST, CODE and DATA/BSS sections of any libraries that it will load at run time. The sizes of these reserved spaces are provided in the header of the executable file. The loader has considerable freedom to place these sections anywhere it can in the event that the reserved spaces are insufficient. The only requirements are that the CONST section of the library function is within a range of $\pm$ 2GB of the CODE section of the library, and the DATA and BSS sections of the library are within $\pm$ 2GB of DATAP. The library function may be compiled with a compiler option that tells it not to use DATAP. The function will load the absolute address of its DATA section into a general purpose register and access its data with this register as pointer.

10.9 Library function dispatch system

Newer versions of Linux have a feature called Gnu indirect function which makes it possible to choose between different versions of a function at load time depending on, for example, the microprocessor version. This feature will not be copied in the ForwardCom system because it relies on a procedure linkage table (PLT). Instead, we can make a dispatcher system to be used with load-time linking. The library can contain a dispatch function which tells which version of a library function to load. The loader will first load the dispatch function and call it. The dispatch function returns the name of the chosen version of the desired function. The loader then unloads the dispatch function and links the chosen function into the main program. The dispatch function must have access to information about the hardware configuration, command line parameters, environment variables, operating system, user interface framework, and anything else that it might need to choose which version of the function to use.

10.10 Predicting the stack size

In most cases, it is possible to calculate exactly how much stack space an application needs. The compiler knows how much stack space it has allocated in each function. We only have to make the compiler save this information. This can be accomplished in the following way. If a function A calls a function B then we want the compiler to save information about the difference between the value of the stack pointer when A is called and the stack pointer when B is called. These values can then be summed up for the whole chain of nested function calls. If function A can call both function B and function C then each branch of the call tree is analyzed and the value for the branch that uses most stack space is used. If a function is compiled separately into its own object file, then the information must be stored in the object file.

A function can use any amount of memory space below the address pointed to by the stack pointer (a so-called red zone) if this is included in the stack size reported in the object file, provided that the system has a separate system stack.

The amount of stack space that a function uses will depend on the maximum vector length if full vectors are saved on the stack. All values for required stack space are linear functions of the vector length: Stack_frame_size = Constant + Factor · Max_vector_length. Thus, there are two values to save for each function and branch: Constant and Factor. We need separate calculations for each thread and possibly also information about the number of threads. If there are two stacks then we need to save separate values for the call stack and the data stack. The size of the call stack does not depend on the maximum vector length.

The linker will add up all this information and store it in the header of the executable file. The maximum vector length is known when the program is loaded, so that the loader can finish the calculations and allocate a stack of the calculated size before the program is loaded. This will prevent stack overflow and fragmentation of the stack memory. Some programs will use as many threads as there are CPU cores, for optimal performance. It is not essential, though, to know how many threads will be created because each stack can be placed anywhere in memory if thread memory protection is used (see page 72).

In theory, it is possible to avoid the need for virtual address translation if the following four conditions are met:

- The required stack size can be predicted and sufficient stack space is allocated when a program is loaded and when additional threads are created.

- Static variables are addressed relative to the data section pointer. Multiple running instances of the same program have different values in the data section pointer.

- The heap manager can handle fragmented physical memory in case of heap overflow.

- There is sufficient memory so that no application needs to be swapped to a hard disk.

A possible alternative to calculating the stack space is to measure the actual stack use the first time a program is run, and then rely on statistics to predict the stack use in subsequent runs. The same method can be used for heap space. This method is simpler, but less reliable. The calculation of stack requirements based on the compiler is sure to cover all branches of a program, while a statistical method will only include branches that have actually been used.

We may implement a hardware register that measures the stack use. This stack-measurement register is updated every time the stack grows. We can reset the stack-measurement register when a program starts and read it when the program finishes. We don't need a hardware register to measure heap size. This information can be retrieved from the heap manager.

These proposals can eliminate or reduce memory fragmentation in many cases so that we only need a small memory map which can be stored on the CPU chip. Each process and each thread will have its own memory map. However, we cannot completely eliminate memory fragmentation and the need for virtual memory translation because of the complications discussed on page 72.

10.11 Exception handling, stack unrolling and debug information

Executable files must contain information about the stack frame of each function for the sake of exception handling and stack unrolling for programming languages that support structured exception handling. It should also be used for programming languages that do not support structured exception handling in order to facilitate stack tracing by a debugger.

This system should be standardized, and both single stack and dual stack systems should be supported. It is recommended to use a table-based method that does not require a stack frame register.

Debuggers need information about line numbers, variable names, etc. This information should be included in object files when requested. The debug information may be copied into the executable file

or saved in a separate file which is stored together with the executable file. It is yet to be decided which system to use.

10.12 Assembly language syntax

The definition of a new instruction set should include the definition of a standardized assembly language syntax. The syntax should be suitable for human processing, not only for machine processing. Mnemonic names should be long enough to make sense. Instructions should have the destination operand first. We must avoid a situation similar to the x86 environment where many different syntaxes are in use, with different instruction names and different orders of the operands.

The assembly code has one instruction on each line, consisting of an instruction mnemonic and its operands. It is proposed to add suffix codes to instruction mnemonics, separated by a dot, to indicate the operand type: 8, 16, 32, 64, 128 for integer operand size, and f, d, q for single, double and quadruple precision floating point operands. Add a 'z' to the integer operand type if the result must be zero-extended into a 64-bit general purpose register. Without the 'z', the assembler will pick the shortest instruction regardless of whether the result may overflow into additional bits. For example, add.16 r0,r0,1 may use a tiny instruction with 64 bit operand type that can overflow beyond 16 bits, while add.16z r0,r0,1 must use an instruction with a 16-bit operand type to make sure that the remaining bits of the general purpose register will be zero.

Memory operands are indicated with square brackets. The vector length specifier of a memory operand is indicated as ", length=register" after the address operand. A mask register is indicated as ", mask=register". Example:

```
add.f v0, v1, [r2+0x100, length=r3], mask=v4
```

This will add the float vector v1 and the vector memory operand with pointer r2, offset 0x100 and length r3 (bytes) and save the result in v0, using mask v4.

An array operand is indicated as "(base register)+(index register)*(scale factor)". If there is a scale factor ($\neq \pm 1$) then the scale factor must match the operand size indicated by the operand type suffix. A limit to the index is indicated with
", limit=value". Example:

```
add.64 r0, r1, [r2+r3*8, limit=999], mask=r4
```

This will load a 64-bit integer from an array of 1000 elements with base address r2 and index r3, where the index is scaled by the operand size (64 bits = 8 bytes), with a limit of r3 $\leq$ 999, add the loaded number with the value of r1 and store the result in r0, using mask r4.

The same instruction may alternatively be written in the style of a function:

```
r0 = add.64 (r1, [r2+r3*8, limit=999], mask=r4)
```

Move instructions may conveniently be written simply with an equal sign, for example:

```
r0 = r1                ; copy general purpose register
r2 = 0xFFFF            ; set general purpose register to constant
v3 = [r4, length=r5] ; read memory operand into vector register
```

Comments are indicated with a semicolon or a double slash.

Traditional assemblers often have metaprogramming features such as macros, preprocessing conditionals and preprocessing loops. The syntaxes used for these features look like awkward ad hoc solutions without no overall logical structure. We would prefer a syntax that makes a clear distinction between metaprogramming and regular assembly code. The metaprogramming syntax should support integer and floating point variables, strings, macros, conditionals and loops in a way that resembles a structured programming language.

Chapter 11

Conclusion

The proposed ForwardCom instruction set architecture is a consistent, modular, flexible, orthogonal, scalable and expansible instruction set offering a good compromise between the RISC principle that gives fast decoding, and the CISC principle that gives more compact code and more work done per instruction. Each instruction can be coded in many different variants with different operand types, different memory addressing modes, scalars, vectors, predicates, masks and option bits. Support for efficient vector processing and out-of-order execution is a basic part of the design rather than a sub-optimal patch added later as we have seen in other systems.

General instructions, such as e. g. addition, can be coded in many different formats with integer operands of different sizes and floating point operands of different precisions. The operands can be scalars or vectors of any length. Operands can be registers, immediate constants, or memory operands with different addressing modes. All in all, the same basic instruction can have many different variants with the same operation code where other instruction sets have many different instructions to cover the same diversity. This simplifies the hardware implementation. The design also has plenty of space for single-format instructions with fewer variants.

The instructions are designed so that the microprocessor pipeline can be simple and efficient. All instructions fit into the same simple and logical template system that will make both hardware and software simpler and more efficient.

The decoder front-end can load multiple instructions per clock cycle because it is easy to detect the length of each instruction, and the decoder needs only distinguish between a few different instruction sizes. Actually, the only instruction size that must be supported is single-word. It is possible to make a working program with only single-word (32 bits) instructions, but it is highly recommended to also support double-word instructions. Triple-word instructions is a convenience that may be supported if it can be implemented without reducing the overall decoding speed. Tiny instructions (two in one code word) are useful for making the code more compact.

It is possible to add support for longer instructions in future extensions, but the priority has been to avoid any bottleneck in the decoding of instruction length (which is a serious bottleneck in the x86 architecture).

The code format is designed to be compact in order to save code cache space. This compactness is obtained in several ways. The same instructions can be coded in different sizes with two- and three-operand forms, different sizes of immediate constants, shifted immediate constants, and relative addresses with different sizes of offsets and scale factors, while avoiding absolute addresses that would require 64 bits for the address alone. It is always possible to choose the smallest version of an instruction that fits the particular need. The load on the data cache can be reduced by storing immediate constants in the code rather than in memory operands.

Most instructions can have a mask register which is used for predication in scalar instructions and masking in vector instructions. The same mask register is also used for specifying various options such as rounding mode, exception handling, etc., that would otherwise require extra bits in the instruction code.

The introduction of vector registers with variable length is an important improvement over the most common current architectures. The ForwardCom vector system has the following advantages:

- The system is scalable. Different microprocessors can have different maximum vector lengths with no upper limit. It can be used for small embedded systems as well as large supercomputers with very long vectors.

- The same code can run on different microprocessors with different maximum vector lengths and automatically utilize the full vector capabilities of each microprocessor.

- The code does not have to be recompiled when a new microprocessor version with longer vectors becomes available. Software developers do not have to maintain multiple versions of their software for different vector lengths.

- The software can save and restore a vector register in a way that is guaranteed to work with future processors with longer vectors. The inability to do so is a big problem in current architectures.

- Only the part of a vector register that is actually used needs to be saved and restored. Each vector register includes information about how many bytes of it are used. Therefore, no unnecessary resources are wasted on saving a full-length vector if it is unused or only partially used.

- A special addressing mode supports a very efficient loop structure that will automatically use the maximum vector length on all but the last iteration of an array loop. The last iteration will automatically use a shorter vector to handle the remaining array elements in case the array size is not divisible by the maximum vector length. There is no need to handle the remaining elements separately outside the main loop and no need to make separate versions of the loop for different special cases.

- Functions can have variable-length vector registers as parameters. This makes it easy for the compiler to vectorize loops that contain function calls.

- Instructions with vector register operands need no extra information about the vector length because this information is included in the vector registers. This makes these instructions more compact. Instructions with vector memory operands do need this extra information, though.

- The system takes into account the special needs of microprocessors with very long vectors where transport delays across a vector may depend on the vector length.

The memory model is flexible with relative addresses. Everything is position-independent. Memory management is simpler than in many current systems with less need for virtual address translation. There is no translation lookaside buffer (TLB) and no memory paging, but a simple on-chip memory map. Problems with stack overflow, memory fragmentation, etc. can be avoided completely in most cases. Task switches will be fast because of the small memory map and because of the efficient mechanism for saving vector registers.

The principle that a fundamental redesign enables us to learn from history and integrate late additions into the basic design also applies to the whole ecosystem of ABI standards, function libraries, compilers, linkers and operating system. By defining not only an instruction set, but also ABI standards, binary file formats, interface library standards, etc. we get the further advantage that different compilers and different programming languages will be compatible with each other. It will be possible to write different parts of a program in different programming languages and to use the same function libraries with all compilers. Even different operating systems will be compatible to some degree. It is not an impossible goal to be able to run the same binary program file in different operating systems.

We have also learned from past mistakes that it is difficult to predict future needs. While the ForwardCom instruction set is intended to be flexible with room for future extensions, we may ask whether

the future will bring needs for new features that are difficult to integrate into our design and standards. The best way to prevent such unforeseen problems is to allow input and suggestions from the entire community of hardware and software developers. It is important that the design and standards are developed through an open process that allows everybody to comment and make suggestions. We have already seen the problems of leaving this to a commercial industry. The industry often makes short-term decisions for marketing reasons. Patents, license restrictions and trade secrets harm competition and prevent niche operators from entering the market. New features and instruction set extensions are kept secret for competitive reasons until it is too late to change them in case the IT community comes up with better proposals.

The ForwardCom project is developed as a contribution to an open development process based on the philosophy that these problems can be avoided through openness and collaboration.

Chapter 12

Revision history

Version 1.03, 2016-08-01.

- Minor changes and additions to manual.

- Three new instructions added.

Version 1.02, 2016-06-25.

- Name changed to ForwardCom.

- Moved to github.

- Various security features added.

- Support for dual stack.

- Some instruction formats modified, including more formats for jump and call instructions.

- System call, system return and trap instructions added.

- New addressing mode for arrays with bounds checking.

- Several instructions modified or added.

- Memory management and ABI standards described in more detail.

- Instruction list in comma separated file instruction_list.csv.

- Object file format defined in file elf_forwardcom.h

Version 1.01, 2016-05-10.

- The instruction set is given the name CRISC1.

- The length of a vector register is stored in the register itself. The basic code structure is modified as a consequence of this. Function calling conventions are also simplified as a consequence of this.

- All user-level instructions are defined.

- The entire text has been rewritten and updated.

Version 1.00, 2016-03-22.

This document is the result of a long discussion on Agner Fog's blog , starting on 2015-12-27, as well as input from the RISC-V mailing list and the Opencores forum.

Additional inspiration was found in various sources listed on page 6.

Version 1.00 of this manual was published at www.agner.org/optimize.

Chapter 13

Copyright notice

This document is copyrighted in 2016 by Agner Fog with a Creative Commons attribution-share alike license. creativecommons.org/licenses/by-sa/4.0/legalcode.