The Hardware Architecture of the CRISP Microprocessor

David R. Ditzel
Hubert R. McLellan

AT&T Bell Laboratories
Murray Hill, N.J. 07974

Alan D. Berenbaum

AT&T Information Systems
Holmdel, N.J. 07733

The CRISP Microprocessor

The AT&T CRISP Microprocessor is a high performance general purpose 32-bit processor. It has been implemented as a single CMOS chip containing 172,163 transistors in a 1.75μ CMOS technology and runs at a clock frequency of 16 MHz.[1] The CRISP Microprocessor achieves performance through traditional techniques, such as pipelining, and from several new techniques not before found in microprocessor designs. This paper focuses on a detailed description of hardware architecture, including the pipeline structure and details of the architectural innovations. A brief introduction to the instruction-set and major features are given for background.

The CRISP instruction-set is carefully streamlined to allow an efficient pipelined implementation. CRISP consists of two logically separate machines, a Prefetch and Decode Unit and an Execution Unit. These units are connected by a decoded-instruction cache. With this decoupled parallel operation and internal pipelining, CRISP is capable of issuing a new instruction every cycle. Fast operand access is accomplished with *Stack Cache* registers[2] instead of general purpose data registers. Efficient procedure calls are possible because of the Stack Cache and a minimal subroutine linkage mechanism. Branches can be executed in zero time by *Branch Folding*.[3] A highly decoded instruction cache allows memory-to-memory style instructions to be often executed in a single cycle by a RISC style Execution Unit. Code generation by compilers is simplified as there are only a few instructions and addressing modes to chose from, and register allocation is not required. A variable-length instruction-encoding yields good code density (equal to the VAX) and reduces off-chip instruction traffic. These instructions are translated by the Prefetch Decode Unit to a fixed-length internal format for high speed execution.

Genesis of the Design

Since 1975 the Bell Labs C Machine Project has designed several computer architectures to support efficiently the C Programming Language.[2,4,5,6] These designs evolved into the current C Machine instruction-set architecture. The CRISP Microprocessor represents a particular implementation of the C Machine architecture. The current architecture stabilized in 1981 for an ECL implementation that was never completed. The design team for CRISP was formed in April 1983 and the first mask was submitted for fabrication in February 1986.

The goal of the C Machine Project was to design and build a computer with significantly better cost/performance characteristics than commercially available computers. We were seeking architectural changes that could provide an order of magnitude greater performance than the machines commonly available to us. The C Machine was designed with an iterative methodology based on extensive measurements of C programs. Part of the method consisted of a cycle of proposing a machine, writing a compiler, running a large body of UNIX software through the compiler and analysis tools, and then using measurements to add or delete features and propose a new machine. These measurements guided the hardware/software tradeoffs made between compiler technology, well known architectural techniques, and hardware limitations.

In particular, these measurements focused our efforts on four areas. First, we reduced the cost of procedure calls, because up to half of the execution time of our VAX programs was consumed by procedure call overhead. Second, we looked for ways to use a large number of registers effectively without placing the burden on complex compiler technology that we might never achieve. Third, we concentrated on simple instructions, streamlining them to permit an efficient pipelined implementation. Fourth, we sought to avoid performance problems caused by pipeline breakage from branches.

Instruction Set Architecture

The CRISP instruction-set contains a small number of instructions and addressing modes. The instruction-set architecture is a registerless, 2½ address memory-to-memory machine. The ½ address refers to being able store the destination of an operation in a single location called the accumulator, which is implemented as the first variable location on the stack frame. Two, one, and zero address instructions are also supported.

CRISP is a full 32-bit machine. The wordsize and addresses are 32-bits. Integer values may be 32-bit words, 16-bit halfwords or 8-bit bytes. Bytes and halfwords are either sign extended or zero filled to 32-bit values before an ALU operation takes place.

CRISP provides four addressing modes per operand for accessing most data. Bytes, halfwords and words can be specified with an absolute address, an offset from a stack pointer, or indirectly though an address specified by an offset from a stack pointer. Immediate constants can be placed in the instruction itself.

ALU-type instructions are a minimal and sufficient set for compiler writers. Operations are addition, subtraction, multiplication, division, logical and arithmetic shifts, and bitwise and, or, and exclusive or. Both 2 address and 2½ address forms are provided for most ALU operations. A compare instruction tests two operands for equality, signed less than, and unsigned less than. Only the compare instruction can set a single condition code flag; no other ALU operations affect it. The flag may be used later by a conditional branch instruction to branch if the flag is true, or if it is false. Conditional branch instructions also contain a single static branch prediction bit, which may be set by a compiler. This bit specifies whether the branch is normally expected to be taken. Four instructions are used for procedure calling: **call** saves the return PC and branches to the subroutine; **enter** allocates space for a new stack frame and flushes Stack Cache entries if necessary; **return** deallocates the stack frame and branches back to the caller; and **catch** restores Stack Cache entries from memory if necessary. A summary of CRISP instructions is shown is Table 1. A more complete description of the instruction-set can be found elsewhere.[7]

Instruction Encoding

The instruction encoding is designed with two primary considerations. First, the instruction length must be easily determined. Therefore, the length is encoded in the first two bits of each instruction. Since all instructions are multiples of two bytes, this unit is referred to as an instruction parcel. Second, static and dynamic code size should be made as small as possible without interfering with performance issues. Instructions that require two 32-bit addresses or operands, can use the five parcel form shown in Figure 1. The three parcel form can be used to provide a single 32-bit operand or two 16-bit operands. The single parcel format has a 5-bit opcode field which defines the most frequent combinations of operations and addressing modes occurring in the three and five parcel forms. This highly encoded single parcel form typically accounts for 80 percent of all instructions. The resulting programs are compact despite only three instruction lengths and a simple instruction-set — program size is about the same as that of the VAX with its multitude of complex modes and instruction lengths.

Architectural Overview

Figure 2 illustrates the basic functional blocks of the CRISP microprocessor. There are three distinct caches, two major data-path blocks and an I/O section to communicate off-chip. These six functional blocks operate autonomously, without any central controller. These units are (roughly in order of instruction flow):

Input/Output. The CRISP I/O is fully synchronous, and can complete an I/O transaction every clock cycle. There are separate address and data busses. The data bus is 32 bits wide, while the address bus provides 30-bit word addresses. Bytes within words are accessed via four byte-mark strobes. Although the I/O can maintain a one transaction-per-cycle rate, this rate is not mandatory. Performance degrades gracefully as wait states are added. In addition, a block transfer mode is provided for systems using nibble-mode or page-mode RAMS, where the first access to a block may take more time than subsequent sequential accesses. The I/O protocol also supports coprocessors, wait states, slow tri-stating peripherals, interlocked bus operations and the parallel connection of multiple CPU chips for fault-tolerant, self-checking operation. The microprocessor is packaged in a 125-pin pin grid array, using 96 active signal pins and 20 power and ground pins.

Prefetch Buffer. The Prefetch Buffer is an instruction cache similar to those found in the Motorola 68020 or AT&T WE32100. The purpose of the Prefetch Buffer is to match the limited bandwidth through the microprocessor data pins to the internal demands of instruction decode and execution. The Prefetch Buffer is implemented as a direct-mapped sector-cache with two valid bits per line. There are 32 lines, each of which contains two double-word blocks, for a total of 512 bytes. Instructions are stored in the cache with the same compact encoding as in main memory. All program text is fetched with double-word block I/O transactions. The Prefetch Buffer can deliver a 64-bit block of encoded instructions to the Prefetch/Decode Unit every cycle.

Prefetch/Decode Unit. The job of the Prefetch and Decode Unit(PDU) job is to take the highly encoded instructions stored in the Prefetch Buffer and decode them into a canonical 192-bit internal instruction that can be efficiently executed by the Execution Unit. Before decoding, the PDU must align instructions from the double-word blocks delivered by the Prefetch Buffer and the I/O. The alignment is achieved by an eight entry, 16-bit parcel instruction queue that emits one to five parcels every cycle. The PDU as a whole can decode and deliver up to two decoded instructions every cycle. Once started, the PDU operates autonomously. It follows the instruction stream, including data-independent branches, retrieves instructions from the Prefetch Buffer, decodes them, and deposits them into the Decoded Instruction Cache.

Decoded Instruction Cache. The Decoded Instruction Cache acts as a buffer between the PDU and the Execution Unit. Like the Prefetch Buffer, it is organized as a direct-mapped cache, with 32 192-bit entries. Each entry is a fully decoded instruction, so that instructions that issue from the Decoded Instruction Cache can be executed without any further sign-extension, field extraction, or decode delay. Because of branch folding, to be described later, each Decoded Instruction Cache entry can hold two instructions. The Decoded Instruction Cache, during each clock cycle, can receive an instruction from the PDU, as well as deliver an instruction to the Execution Unit. Like the Prefetch Buffer, the Decoded Instruction Cache can also bypass data directly from the PDU to the Execution Unit.

Execution Unit. The Execution Unit (EU) is optimized for high speed execution, and in some ways resembles RISC machines such as the IBM 801.[8] It consists of three pipeline stages, with a straightforward sequence of operand fetch, ALU operation, then register update. Because CRISP is a memory-to-memory architecture, the EU can calculate addresses, fetch data and align and sign-extend two operands simultaneously. Although most instructions flow through the pipeline in three cycles, for a net rate of one instruction per cycle, more complex instructions such as multiply and divide can take multiple cycles. With the help of branch folding, the peak execution rate can be as high as two instructions every cycle. The 192-bit decoded instruction resembles horizontal microcode, and like a typical microprogrammed machine the EU sees only fixed length instructions. This simplifies next address calculation and makes it easier to issue new instructions every cycle. Unlike other RISC machines, CRISP's internal instruction width is not limited to 32-bits, and unlike CISC machines CRISP does not require a large, static microprogram ROM. CRISP takes instructions that are compact and easy for a compiler to generate and dynamically transforms them into easy to execute RISC instructions.

Stack Cache. Unlike most computers CRISP has no visible data or address registers. Instead, 32 internal Stack Cache registers are mapped into the address space corresponding to the top of the stack. The Stack Cache is byte-addressable and looks like memory in every way. It can contain strings, structures and arrays. The Stack Cache is implemented as a circular buffer of registers, maintained by a head and tail pointer, called the Stack Pointer (SP) and Maximum Stack Pointer (MSP). Memory references that fall between the bounds of the SP and MSP are address the on-chip Stack Cache registers. Management of the Stack Cache is controlled with the **enter** and **catch** instructions during procedure calling. The Stack Cache registers are implemented with two 32-entry, 32-bit wide memories. During every cycle the Stack Cache performs two distinct reads and a single write.

Pipeline Structure and Terminology

Pipeline stages are clocked once each cycle. Each clock cycle is subdivided into four equal duration phases, numbered 1 to 4. Pipeline registers are built from a master/slave register pair. The master registers are always clocked during phase 4, whereas the slave registers are conditionally clocked during the following phase 1. Registers are implemented with transparent latches.

To balance the amount of computation in each pipeline stage, assumptions were made about the time required for various functions. A 32-bit addition and an on-chip memory access are roughly comparable and take approximately half a clock cycle. An ALU operation including shifting, sign extension, and the result alignment was allocated a full clock cycle. The cache memories are sequenced twice each clock cycle. They are read in the first half of a cycle, and written in the second half of a cycle. These constraints define the clock cycle time of the machine.

We will use the notion *<pipe-stage>.<register-name>* to identify particular pipeline registers. For example, PDR.PC is the PC register of the PDR pipeline stage. Particular bits of a register are described by following the name of the register with the selected bits enclosed in angle brackets. Thus, x<2:6> specifies bits 2 through 6 of the x field. The names of busses or functional units are shown in italics, the names of explicit instructions are shown in bold face.

The Prefetch and Decode Unit

The PDU is a 3-stage pipeline responsible for the fetching instructions from external memory and then decoding and storing them in the Decoded Instruction Cache. Externally, instructions are either one, three or five parcels in length. The PDU translates these variable length instructions into a single 192-bit canonical internal form. Once started, the PDU fetches instructions on its own, decoding and following branches and calls in the instruction stream. This autonomous action is only stopped by a demand request from the EU, or an inability to continue following the instruction path. This might occur when the address of the next instruction is data-dependent as in a procedure return or indirect jump.

The PDU's pipeline stages will now be described in the order the instructions flow through them. The PDU's three pipeline stages are: the Prefetch Buffer Register (PBR), the Prefetch Decode Register (PDR), and the Prefetch Instruction Register (PIR). A simplified schematic of the PDU is shown in Figure 3.

PBR Stage: The PBR stage is the head of the PDU pipeline and consists of a single register, the PBR.PC. The PBR.PC addresses the Prefetch Buffer and performs a blind prefetch of 64-bit instruction blocks. During the first half of each cycle, the Prefetch Buffer is indexed by the low-order bits of PBR.PC and the four instruction parcels retrieved are placed in the W, X, Y, and Z registers. If the high-order bits of the PBR.PC match the tag associated with this entry, these four parcels are valid and may be loaded into the Instruction Queue during phase 4 of the same cycle. When there is space in the Instruction Queue for these four parcels, the PBR.PC is incremented and the next Prefetch Buffer entry is accessed. This sequence continues every cycle until an instruction discontinuity occurs. A new PBR.PC is then loaded and a new instruction stream is started.

The Prefetch Buffer is a direct-mapped sector-cache, containing 64 64-bit sub-blocks and 32 22-bit tags, each with 2 valid bits and 1 execution level privilege bit per tag. The two valid bits implement a four-word line comprised of two 64-bit entries. These bits define which of the two entries are valid. The Prefetch Buffer Tag memory contains an integral comparator to determine a cache hit/miss. If a miss occurs in the first block of a four-word cache line, two requests for double-word block transfers are made to the I/O, at the memory address specified by the PBR.PC. If the miss occurs in the second block, only one double-word I/O request is made. If the PDU is waiting for an instruction because of a Prefetch Buffer cache miss, when a double-word arrives from the I/O at the input side of the Prefetch Buffer, it is written both into the cache and the W, X, Y, and Z registers. This bypassing capability saves a clock cycle instead of having to wait for the data to be written into the Prefetch Buffer first.

There are two advantages to this cache organization. If the PBR.PC points to the second double-word (as it might after an instruction discontinuity), it can be fetched without having to wait for the entire four word line. Also, system performance is improved by allowing other I/O activity to intervene during instruction prefetches. Block transfers of greater than two words were found to degrade performance by stalling EU execution.

PDR Stage: The PDR stage consists of an eight parcel Instruction Queue, PDR.QA through PDR.QH, and the PDR.PC, which contains the address of the QA parcel. The eight-entry Queue aligns the variable length instructions for the benefit of the next pipeline stage. Five parcels, QA through QE

are available as outputs from the Queue.

An instruction is decoded from the Queue each cycle. After decoding, the one to five parcels comprising that instruction are removed from the Queue. Any remaining valid parcels in the Queue are shifted to fill the vacated entries. Finally, if there is room, the Queue is loaded with the four W, X, Y, and Z parcels from the Prefetch Buffer.

PIR Stage: The PIR Stage contains a completely decoded instruction, composed of six 32-bit fields. These fields are: the PC of the instruction, the Next-PC address, the Alternate Next-PC address, a constant or address for the left and right operands, and 32-bits of control.

At the beginning of a cycle, the QA parcel is fed into a PLA, the PDUPLA, for decoding. The PDUPLA detects unimplemented or illegal instruction and generates the control signals for the PIR.control field. The PIR.control register contains 32 relatively independent bits containing decoded opcode, addressing mode, and other control information to be used in the EU. The control field contains 5-bits for an internal opcode; the Execution Unit only has to deal with a simple instruction set of 25 instructions. Instructions of different lengths with the same function are translated into a single decoded instruction. For example, one, three and five parcel encodings of an add instruction will all be translated to the same internal opcode, whose bit encoding is different from the external form. The external encodings are selected for the best code compaction, whereas the internal encodings are selected for speed of execution by the EU.

The PIR.LOP and PIR.ROP registers hold the left and right operands of a specified operation. (The left operand field is the combination source/destination field for two-address operations, the right operand field is the source and also holds the operand for monadic operations.) The operands are sign extended 32-bit values of either constants, offsets or absolute addresses extracted from the QA-QE registers of the PDR. The PDUPLA also generates signals to control the multiplexors sourcing the PIR.LOP and PIR.ROP registers. The PDUPLA uses the high order bit of the B and C parcels to sign extend 16-bit constants with the PIR.LOP and PIR.ROP multiplexors

The address of the newly decoded instruction is copied from the PDR.PC to the PIR.PC. The PIR.PC serves as the tag in the decoded instruction cache.

Like microinstructions in some microcoded engines, a decoded CRISP instruction contains an explicit "next address" field, the PIR.Next-PC. Unlike these other machines, this next address field is dynamically generated, rather than being stored with each instruction in a control memory. For non-branching instructions, the next address is calculated by adding the address of the current instruction, the PDR.PC, to the length of the current instruction using the sequential PC adder, *spcadd*. For branching instructions, the next address can be specified either by a 32-bit absolute address in the instruction, or by a PC relative offset.

This organization leads to a significant optimization. Since every decoded instruction contains a next address field, every decoded instruction is capable of branching to any other instruction. Since every instruction is therefore a branch, there is no need to execute separate branch instructions. Whenever a non-branching instruction is followed immediately by a branch, the two are *folded* together to form a single new decoded instruction. To fold branches, CRISP decodes two instructions simultaneously. When a non-branching instruction is followed by a branch, the address of the branch target is used as the next address of the folded pair. This technique is called *Branch Folding*.

Branch Folding can be applied to conditional as well as unconditional branches. Unconditional branches require only a single Next-PC field, namely the target address of the branch. Conditional branches additionally require a sequential address field, as the true outcome of the branch can not be decided until the instruction reaches the Execution Unit. A second field, called the Alternate Next-PC holds this second address.

Forming the Next-PC field for a folded instruction is more difficult than for a non-folded instruction. The main problem is that PC relative branches are relative to their own addresses, not the address of the instruction in the PDR.PC. To compensate, the length of the non-branch instruction being folded must be added to the branch offset before it is added to the PDR.PC. CRISP only folds one parcel branches with one and three parcel non-branching instructions. This decision was based on measurements that showed that about 95% of all branches executed used the one parcel instruction format. The proper 10-bit branch offset is selected by the *tpc* multiplexor and added to the length of the instruction preceding the branch by the *tpinc* adder. The next address is then obtained by adding this adjusted offset to the PDR.PC with the *tpcadd* adder. Separate non-folded branches are also allowed for the infrequently occurring case when folding is not permitted. For this case, a PC relative next address is obtained by adding the 10-bit offset from the PDR.QA parcel to a branch offset of zero, then adding to the PDR.PC with the *tpcadd* adder.

If the instruction being decoded is a branch that is predicted to be taken, the Next-PC will be selected from the branch target *tpc*, or from the 32-bit address in the QB and QC parcels. Otherwise, the Next-PC will be the next sequential instruction address via the *spc* bus. The Alternate-PC value will select the alternative not selected by the Next-PC.

The PDU follows the path of branches and calls, and the predicted path of conditional branches. The flow of control in the PDU is directed by the instruction being decoded in the PDR stage. For a three parcel branch or call, the target address is brought from the QB and QC parcels into the PBR.PC and PDR.PC. For PC relative and folded branches the target address is brought to the PBR.PC via the *tpc* bus.

As mentioned before, the PDU may be stopped from its prefetching by the EU to do a demand fetch. The EU redirects the PDU with a new PBR.PC and PDR.PC from the *eunpc* bus. Often the PDU is only slightly behind the current execution of the EU, and may already be fetching and decoding the instruction requested by the EU. Rather than throw away this useful work by completely restarting the PDU, two comparators, the *pdeq* and *pieq*, check to see if the requested instruction is already in the pipeline. If so, the PDU ignores the request and continues prefetching.

The Decoded Instruction Cache

Decoded instructions waiting in the PDU's PIR stage are written into the Decoded Instruction Cache in the second half of a clock cycle. During the first half of a clock cycle decoded instructions are read by the Execution Unit. The output of the Decoded Instruction Cache is latched every cycle into a temporary register at the end of phase 2. If the EU is waiting for an instruction because of a Decoded Instruction Cache miss, the instruction will be bypassed to the EU through the Decoded Instruction Cache directly from the PDU's PIR stage. This optimization saves one cycle latency in recovering from a Decoded Instruction Cache miss.

The Execution Unit

The Execution Unit is responsible for the execution of instructions from the Decoded Instruction Cache. The EU is an autonomous unit and only communicates with the PDU when it can no longer proceed because of a Decoded Instruction Cache miss. The EU contains a few processor registers dedicated to specific functions. These are the Program Status Word (PSW), Stack Pointer (SP), Maximum Stack Pointer (MSP), Interrupt Stack Pointer (ISP), Vector Base (VB) register, and Timer. Only one of the SP or ISP is enabled at any one time and it is referred to as the current stack pointer. The Timer register can be configured to count either clock ticks, or instructions executed. The SP, MSP, ISP and VB are aligned on four-word boundaries. This alignment allows indexing within four words of these register values without the use of a full 32-bit adder. An address is formed by or'ing into the low order bits.

The EU has three major pipeline stages, as shown in the simplified schematic of Figure 4. Instructions flow from the Instruction Register (IR) stage to the Operand Register (OR) stage and then to the Result Register (RR) stage. Although most instructions can pass from one stage to the next in one cycle, some instructions, such as multiply, can take multiple cycles. Indirect memory references and some data hazards can also cause instructions to take more than one cycle in a single pipeline stage. Each pipeline stage has a field for the instruction address (PC) — there is no explicit "PC" register. A PC for every instruction in the pipeline is necessary for the precise handling of faults and interrupts.

IR Stage: The IR stage holds the addresses of the operands of an instruction, so the addresses of all data fetches come from the IR. It holds an operand's value if it is an immediate constant. For sequencing control the IR stage also contains the PC of the instruction, the Next-PC, and Alternate-PC. When the IR is ready to be loaded it fetches the next instruction from the Decoded Instruction Cache. If the IR.Next-PC is the same as the tag, a Decoded Instruction Cache hit occurs and the entry is marked valid, by setting the IR.valid bit. If there is not a hit, the IR.valid bit will be turned off to invalidate the pipeline stage. If an operand is an immediate constant, or its addressing mode is absolute, its data is copied directly from the Decoded Instruction Cache into the left operand or right operand fields of the IR. If its addressing mode is stack-relative, the data in the operand field of the Decoded Instruction Cache is used as an offset and added to the Stack Pointer before it is loaded into the IR. Two 28-bit adders allow the simultaneous calculation of left and right operand addresses. The SP adders operate on only the top 28-bits of the operand offset and SP. As the SP is aligned on four word boundaries, its four low order bits are zero, and the low four bits of the operand address are simply copied to the low four bits of the operand address to complete the full 32-bit address.

If an indirect memory reference is selected, then the address of the operand must first be fetched. Since the IR stage is responsible for operand addresses, the indirect fetch of an operand address from the IR.rop or IR.lop is returned back to the IR.rop or IR.lop, respectively. After the indirect fetch is completed, the addressing mode for that operand is changed to eliminate indirection and normal execution resumes. The pointer being fetched for an indirect operation may be located in either the Stack Cache or in off-chip memory. If the operand is located off-chip, it is returned to the IR.lop or IR.rop via the *data_in* bus. If the operand is located in the Stack Cache, it is returned to the IR.lop via the left stack cache bus, or to the IR.rop via the right stack cache bus. Indirect jumps are treated similarly except that the data is returned to the IR.Next-PC field instead of the IR.rop.

In the first half of every cycle address bits <2:6> from the IR stage's left and right operand fields are sent to the Stack Cache and used to perform two simultaneous reads from the memory. At the same time, range check circuitry checks the full virtual addresses to see if they reference operands in the Stack Cache. If an address is greater than or equal to the SP and less than the MSP, then data is returned from the Stack Cache. If this condition is not met, then data must be fetched from off chip. We note that this particular implementation of the Stack cache is essentially the same as that proposed for an earlier C Machine,[2] but in CRISP the addition of the SP to an offset is done in the EU rather than in the PDU.

Pipelining introduces the potential for a Read after Write (RAW) hazard.[9] If the operand address to be fetched by the IR.lop or IR.rop is to be written by an instruction one to two instructions ahead of the instruction in that IR, then the correct value must be bypassed back to the IR stage. The bypassed operand may come from the output of the ALU result on the *alubd* bus (one stage bypassing), or from the output of the RR stage result register on the *rra* bus (two stage bypassing).

OR Stage: The OR stage holds the actual operands for an instruction, rather than the addresses as in the IR stage. If the operand is a constant, the value is copied directly from the IR stage operand registers. If memory was referenced in the IR, then the data may come from the Stack Cache or from off-chip. Since read after write hazards may also occur in fetching data for the OR stage, one and two stage bypassing are also possible sources for the OR stage operand registers. Finally, the contents of the machine registers, such as the SP and PSW, may be a source for the OR stage operand registers.

The detection of RAW hazards is accomplished with four comparators, shown in Figure 4 near the OR.destination and RR.destination address registers. Partial word hazards may be resolved with bypassing only if the data being read is no longer than the data being written and the alignments are compatible. Partial word hazards that do not meet this condition are resolved by stalling the instruction in the IR stage until the data can be correctly read.

The destination address for instructions that do writes is generated in the OR.dest register. For 2-address instructions, the destination address is the same as the IR.lop address. For the **call** instruction, the PC return address is stored at the address of the current stack pointer, obtained via the spbus. For 2½-address instructions, the destination address is that of the accumulator. The accumulator is always one word above that pointed to by the current stack pointer. The address for the accumulator is generated by taking the current stack pointer value and or'ing a one into the address bit 2 position. The ability to add four to the current stack pointer by the use of a single or gate is one of the benefits of quad-word alignment of the stack pointer.

RR stage: The RR result register contains the result of the ALU operation. To perform an ALU operation, operands from the OR.lop and OR.rop stage are first aligned to extract the proper byte, half-word or word. After alignment, the values are sign extended according to whether the values were specified to be signed or unsigned. The ALU always operates on full 32-bit values. Byte and halfword values are not special data types so much as they are ways of compactly storing a representation of a 32-bit value. After the ALU operation is completed, the 32-

bit ALU output is realigned for proper storage in memory or the Stack Cache without further shifting. Writes to the Stack Cache or memory occur during the following cycle.

Branch Control in the EU. The control point for the entire machine is the IR stage's Next-PC register. The IR.Next-PC is used to index the Decoded Instruction Cache to read a new instruction into the IR. When a new instruction is read, the IR.Next-PC may come from the cache's Next-PC or Alternate-PC field. For conditional branches, the Alternate-PC is kept for each pipeline stage in case the branch prediction was wrong. When an incorrectly predicted branch is discovered, the Alternate-PC is placed in the IR.Next-PC and instructions in the pipeline that have followed the wrong path are invalidated.

Pipeline Control

In most cases, instructions flow from one pipeline stage to the next whenever the next stage is *ready*. A pipeline stage is ready if it does not contain a valid instruction (the VALID bit is not set), or if all its data has arrived and the following stage is ready to accept it. If a pipeline stage is busy and the subsequent stage is ready, no-op instructions will be propagated down the pipeline. While the stage is busy, instructions will be held up in previous stages. This back-up stops at the Decoded Instruction Cache: even though the IR may be busy, the Decoded Instruction Cache accepts new instructions from the PDU. The cache decouples the two units and allows them to run autonomously.

Multi-cycle instructions occupy the OR stage of the pipeline, although they may keep the entire EU busy. Shifts and multiplies sit in the OR until they complete, sending no-op instructions down to the RR, but allowing the next instruction to enter the IR. Divides and any sequence which requires I/O (such as catch and enter) require all three pipe stages; when the EU's Sequence PLA is on for these sequences the IR is automatically marked busy. When the Sequence PLA completes, it releases the stages it was holding and goes into an idle state, allowing normal control to resume.

Opcodes in the fully decoded instructions represent classes of external instructions, so there is not a one-to-one match between internal and external opcodes. For example, there is only one internal **add** opcode. There are separate control bits to indicate accumulator destinations and interlocked operations. As another example, the interlock bit, used to specify semaphore operations, could be applied to any instruction, but the PDU only decodes it for **add, and** and **or**. Several control bits are used to speed decoding. One bit tells the control logic that the current instruction can set the condition code, although this information could be obtained by decoding the internal opcode. Additional bits control side effects. If the VALID bit is not set the pipe stage is considered empty. Therefore, when it is necessary to cancel an instruction, it is sufficient to clear the VALID bit in the appropriate pipe stage. If the IS_STORE bit is set, the instruction will do a store when it reaches the bottom of the pipeline. By turning off the IS_STORE bit, it is possible to mark an instruction's presence but preventing it from modifying memory.

Architectural Evaluation

Although the CRISP pipeline is capable of issuing one instruction every cycle (greater than one if folded branches are counted), large programs will usually not attain that rate. Many different events can stall the pipeline, although the precise effects will be different for different programs. With the CRISP implementation of the C Machine, the principal delay comes from Prefetch Buffer cache misses, because the Prefetch Buffer cache is currently relatively small. (This was an intentional tradeoff so that CRISP could provide good performance in systems without an off-chip cache.) Other major delays include fetching operands from off chip, indirect operand address resolution, incorrectly predicted branches (which cause partially completed instructions to be cancelled), and Stack Cache filling and flushing.

Because the biggest contributor to performance loss is the delay in the PDU, program performance depends to a large degree on the size and behavior of the program. If the Prefetch Buffer hit rate is sufficiently high the PDU can keep up with the EU, and instructions will issue in the EU at close to a one-per-cycle rate. Larger programs will have proportionately larger miss rates and hence a slower instruction issue rate.

Table 2 illustrates the instruction issue rate of three different programs. The first is a small synthetic benchmark, about 2K bytes of text, and executes about 14,000 instructions. The second is a Unix program (sed, the stream editor) of about 15K bytes of text, which executes about 25,000 instructions. The third is a large Unix program (the C compiler), of about 80K bytes and executes about 200,000 instructions.

Factor	Benchmark 1	Sed	C Compiler
Basic Instruction	1.0	1.0	1.0
PDU	0.6	0.75	1.4
Data Fetch	0.3	0.5	0.5
Indirect Operands	0.2	0.3	0.3
Branch Miss	0.05	0.2	0.1
Data Stores	0.1	0.15	0.1
Stack Fill/Flush	0.2	0	0.2
Folded Branches	-0.5	-0.9	-0.7
Miscellaneous	0.05	0.05	0.3
Totals	2.0	2.15	2.9

Table 2. Breakdown of instruction delay in cycles/instruction.

All instructions take at least one cycle for the basic instruction. Misses in the EU requiring that the PDU fetch a new instruction contributed another 0.6 to 1.4 cycles for every instruction. Folded branches have a negative contribution in terms of cycles per instruction, since the instructions are executed in zero time.

Many features were evaluated in this manner to determine if they should be included in the final implementation of the CRISP Microprocessor. For example, removing the one-stage bypass paths would add about 0.25 to 0.5 cycles per instruction, while removing the two-stage bypass would add about 0.1 cycles per instruction. The current CRISP implementation folds only one-parcel jumps, and only with one or three parcel instructions. This strategy successfully folds 90 to 95% of all branches.

Performance Evaluation

The bottom line in CPU performance is the length of time required to execute a user's program. This is usually factored into three terms: the total number of instructions required, the average number of cycles per instruction, and the clock rate. With CRISP's 2½ address memory-to-memory instruction-set, the number of instructions executed more closely resembles that of a CISC computer like a VAX than that of a load/store RISC machine. When Branch Folding is included, the CRISP Execution Unit needs to execute fewer instructions than a machine with an extensive instruction set such as the VAX. The combination of pipelining, and integrated caches helps reduce the

average number of clocks per instruction. The simple instruction set contributes to a high clock rate for a pipelined implementation. The CRISP architecture was specifically designed to be cache-memory intensive. With better VLSI technology, a higher performance migration path is easily realized by increasing the size of on-chip caches. In particular, from Table 2, we can see that increasing the size of the the Prefetch Buffer and Stack Cache might reduce the number of cycles/instruction to as low as 2.9-(1.4+.2) = 1.3 cycles/instruction for the C compiler.

Branch Folding allows most branches to execute in zero time. Evaluation and a more detailed description of Branch Folding can be found elsewhere.[3]

Procedure Call overhead is low. Procedure calling is implemented with four instructions: **call** saves the return PC and branches to the subroutine; **enter** allocates space for a new stack frame and flushes Stack Cache entries if necessary; **return** deallocates the stack frame and branches back to the caller; and **catch** restores Stack Cache entries from memory if necessary. When no Stack Cache registers need to be saved and instructions are in the on-chip caches, these four instructions can be executed in a total of only five clock cycles.

Stack Cache registers greatly speed operand access and reduce off-chip memory traffic. Measurements on large programs, such as the C compiler, show 80% of all data references are resolved by the Stack Cache.

Our original performance goal was to provide at least an order of magnitude better performance than a VAX-11/750. At 16 MHz with no wait states, CRISP achieves a performance rating of 13,560 Dhrystones, compared to 997 for a VAX-11/750. This benchmark shows CRISP to be 13.6 times faster.

Performance for several other benchmarks is shown in Table 3. Comparison is made with the DEC VAX-11/780 and with the MIPS Computer Systems R2000 processor as implemented in the M/500 Development System. The figures for CRISP show a system running at 16 MHz with no wait states. A second mask of CRISP using the same technology is expected to run above 20 MHz. The numbers show CRISP to be substantially faster than the VAX, and slightly faster than the MIPS R2000.

For smaller systems CRISP provides a distinct advantage in having the caches on-chip. A one wait-state system can be built at 16 Mhz interfacing directly to dynamic memory with a loss in performance of only about 20% compared to a zero wait-state system. CRISP saves substantially in board area and parts cost being a complete one chip CPU, compared to the R2000 which requires an implementation using two external caches and various support chips, for a total of about 30 chips. CRISP's good code density is also important in some applications where memory costs are still a large portion of system costs, compared to the poorer code density typically found in load/store RISC machines.

Benchmark	VAX-780	R2000	CRISP	CRISP/VAX	CRISP/R2000
ackerman	20.9 sec	1.6 sec	1.1 sec	19.0	1.5
word count	55.0 sec	5.2 sec	4.2 sec	13.1	1.2
quicksort	36.2 sec	4.0 sec	3.4 sec	10.6	1.2
tty driver	17.4 sec	2.2 sec	1.2 sec	14.5	1.8
symbol table	14.6 sec	1.3 sec	1.2 sec	12.2	1.1
buffer release	9.9 sec	0.9 sec	0.8 sec	12.4	1.1
arithmetic	12.8 sec	2.7 sec	1.6 sec	8.0	1.7

Table 3. Relative performance of a 16 MHz CRISP.

Conclusion

The CRISP Microprocessor combines several new architectural techniques into a single design. A highly Decoded Instruction Cache facilitates Branch Folding, reduces pipeline breakage problems caused by branches, and allows peak one cycle execution for fixed length instructions originally generated from a variable length instruction set. Branch Folding gives the appearance that branches can be executed in zero time. The use of a Stack Cache gives efficient use of registers without resorting to complex compiler technology. The Stack Cache substantially reduces procedure call overhead compared to general register approaches, leading to fast procedure calls. These techniques have been implemented in a working, highly pipelined microprocessor.

Acknowledgements

We wish to thank Brian Colbry, Don Freeman, Fred Heaton, George Janac, Kerry Maletsky, Kevin O'Connor, and Shoji for their assistance in the design and implementation of CRISP. Additionally we wish to give our sincere thanks to the many others who supported and contributed to CRISP, particularly those who worked on the 3B-40 and HAWK implementations.

References

1. A. D. Berenbaum, B. W. Colbry, D. R. Ditzel, R. D. Freeman, H. R. McLellan, K. J. O'Connor, and M. Shoji, "A Pipelined 32b Microprocessor with 13Kb of Cache Memory," *Proceedings of the 1987 International Solid State Circuits Conference*, pp. 34-35 (February, 1987).
2. D. R. Ditzel and H. R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *Proc. of Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, pp. 48-56 (March 1982).
3. D. R. Ditzel and H. R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," *Proceedings of the 14th Annual Symposium on Computer Architecture* (June 3-5, 1987).
4. A. G. Fraser, "An Introduction to the C-Machine," *Proceedings of the BTL/WE Microcomputer Symposium*, pp. 9-1 to 9-7 (November 1977).
5. S. C. Johnson, "A 32-Bit Processor Design," Computing Science Technical Report No. 80 (April 1979).
6. D. R. Ditzel and D. A. Patterson, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News* **8**(7) (1980).
7. A. Berenbaum, D. Ditzel, and R. McLellan, *Introduction to the CRISP Instruction-Set Architecture,* Proceedings of the Spring COMPCON (February, 1987), pp. 86-90.
8. G. Radin, "The 801 Minicomputer," *Proceedings of the Symposium on Architectural Support for Programming Languages in Operating Systems*, Palo Alto, CA, pp. 39-47 (March 1982).
9. P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Publisher (1981).

Table 1. CRISP Instructions

Address Type	Name	Function	Addressing Modes	Data Types
2 and 2½ address	add	addition	Immediate	unsigned byte
	sub	subtraction		
	mul	multiplication		
	quo	division		
	and	bitwise and	Absolute	signed byte
	or	bitwise or		
	xor	bitwise exclusive or		
	shr	arithmetic shift right	Stack	unsigned
	ushl	unsigned shift left	Offset	halfword
	ushr	unsigned shift right		
2 address	umul	unsigned multiply		
	uquo	unsigned divide	Stack	
	urem	unsigned remainder	Offset	signed
	cmp.=	equality comparison	Indirect	halfword
	cmp.s<	signed less than comparison		
	cmp.u<	unsigned less than comparison		
	move	move		word
	mova	move effective address		
	addi	bitwise add interlocked		
	andi	bitwise and interlocked		
	ori	bitwise or interlocked		
1 address	jmp	unconditional jump	Absolute	
	ifTjmp	conditional jump if True	Absolute Indirect	
	ifFjmp	conditional jump if False		
	ifCjmp	conditional jump if Carry	Stack Offset Indirect	
	ifOjmp	conditional jump if Overflow		
	call	procedure call	PC Relative	
	kcall	kernel call		
	enter	allocate new stack space		
	return	de-allocate space and return	Immediate	
	catch	restore stack cache		
0 address	nop	no operation		
	cpu	internal register access		
	kret	kernel return		

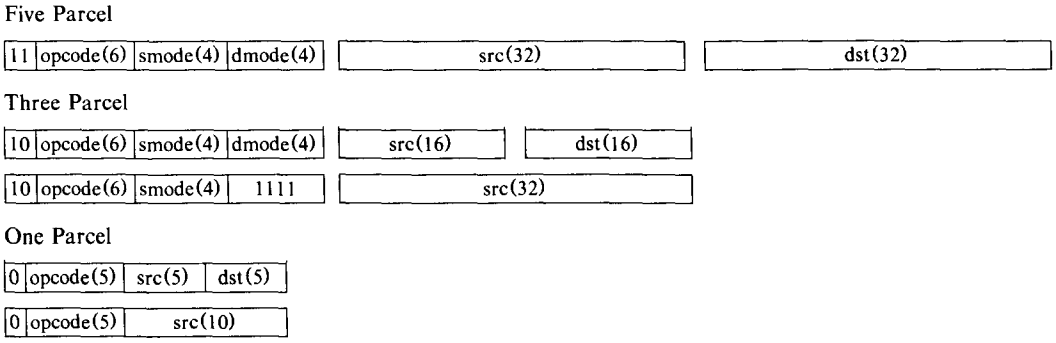

Figure 1. Instruction Encoding Formats.

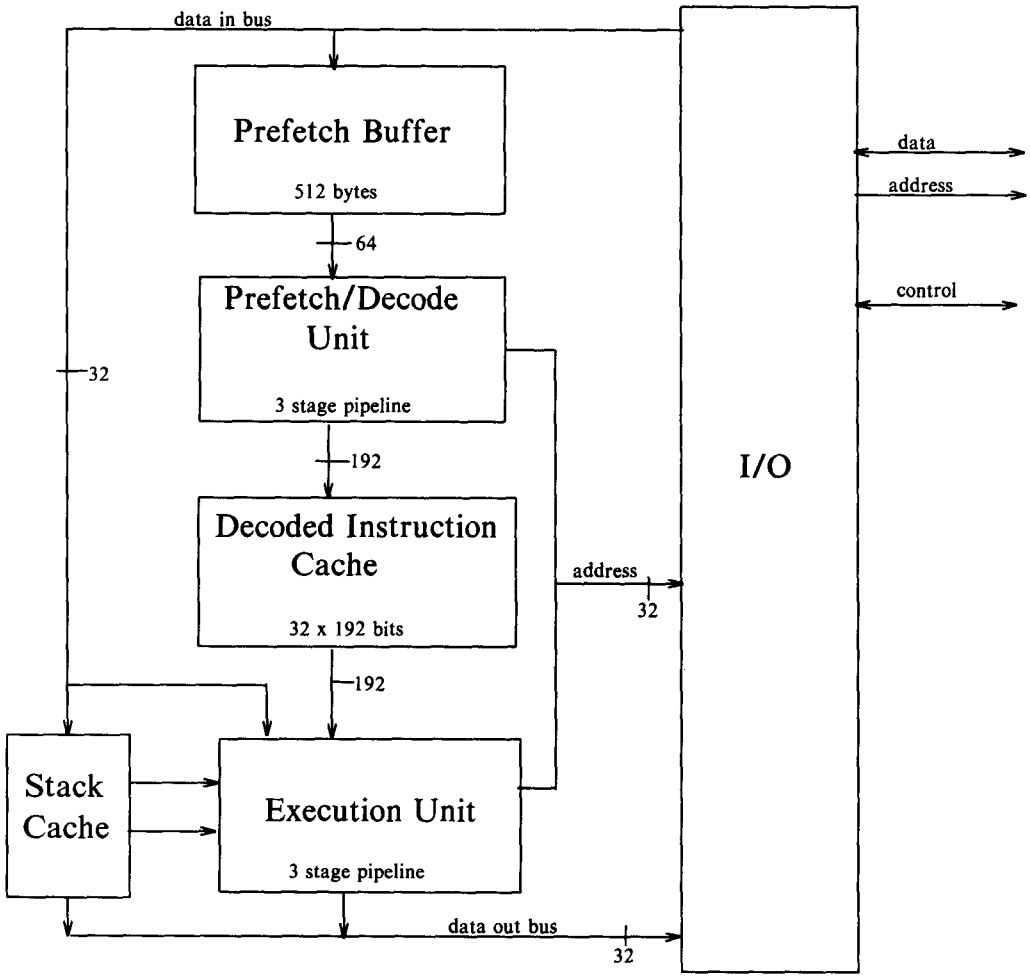

Figure 2. CRISP Microprocessor Block Diagram

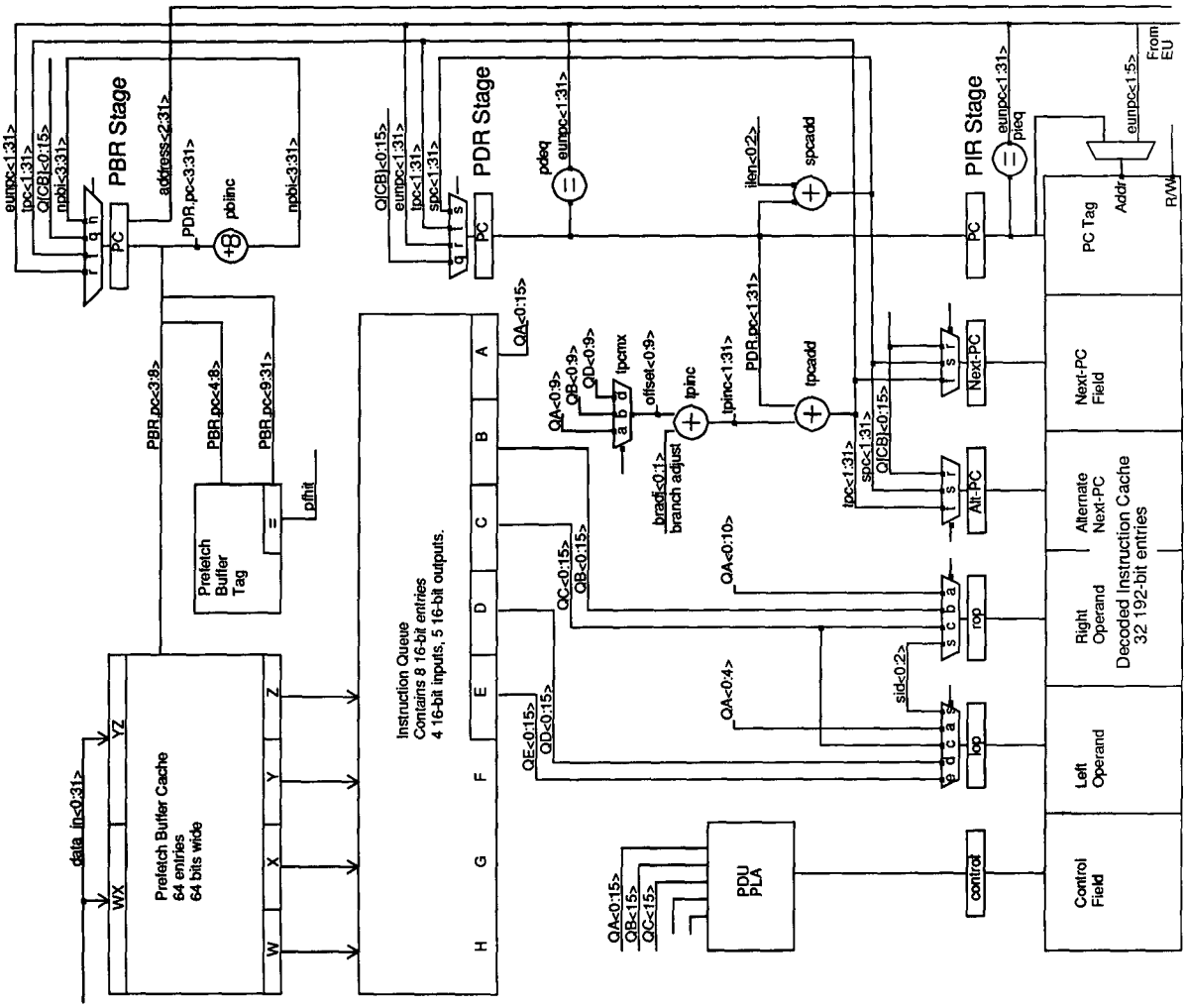

Figure 3. Prefetch and Decode Unit

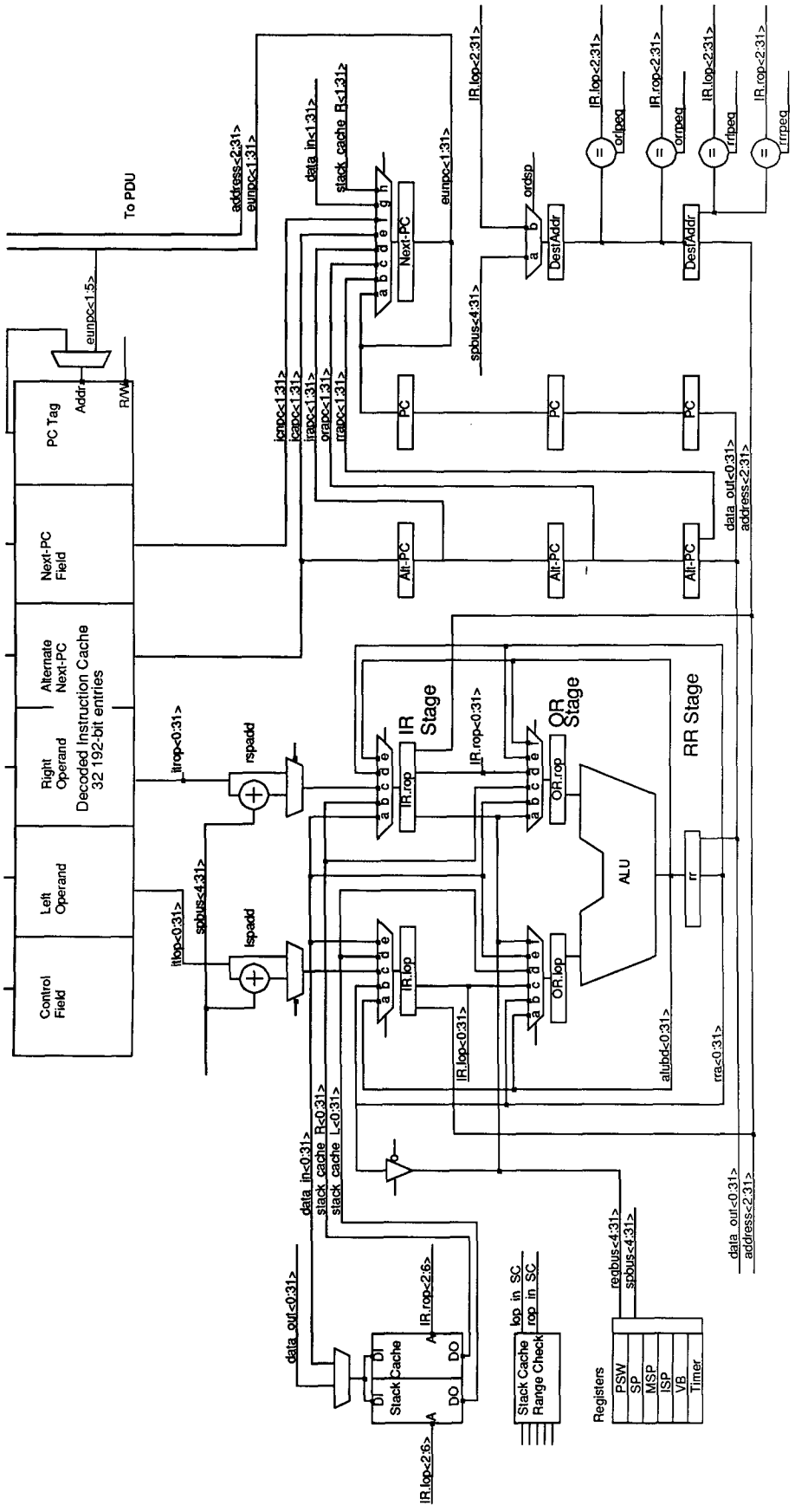

Figure 4. Execution Unit