Architectural Innovations in the CRISP Microprocessor

A. D. Berenbaum

AT&T Information Systems
Holmdel, New Jersey 07733

D. R. Ditzel
H. R. McLellan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

The CRISP Microprocessor

The AT&T CRISP Microprocessor is a high performance general purpose 32-bit processor. It is a single CMOS chip containing 172,163 transistors. The instruction-set architecture is a registerless 2½ address memory-to-memory machine with a small number of instructions and addressing modes.[1] The CRISP instruction-set is relatively independent of any particular implementation. High performance is achieved by pipelining, caches, an efficient instruction-set, and several new architectural techniques. This paper describes some of the innovative aspects of CRISP's hardware architecture.

Architectural Overview

Figure 1 illustrates the basic functional units of the CRISP microprocessor. There are three distinct caches, two major data path blocks, and I/O to communicate off-chip. These six functional blocks operate independently, without any central controller. These units are (roughly in order of execution):

- **Input/Output.** The CRISP I/O is a simple, fast and complete design. It 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, 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. Wait states can be inserted to extend a transaction to any number of clock cycles. In addition, early indication of block mode of operation is provided for systems using nibble-mode or page-mode RAMS, where the first access to a block may take more cycles than subsequent sequential accesses. The I/O protocol also supports coprocessors, slow peripherals, interlocked bus operations and the ability to wire multiple CPU chips in parallel for 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 a traditional instruction cache similar to those found on 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 on instruction decode and execution. The Prefetch Buffer cache is direct-mapped, and is organized as 32 lines, each containing two double-word blocks, for a total of 512 bytes. Instructions are stored in the cache in the same compact encoded form as they are in main memory. All program text is fetched with two-word block I/O accesses. The Prefetch Buffer delivers a 64-bit block to the Prefetch/Decode Unit every cycle.
- **Prefetch and Decode Unit.** The Prefetch and Decode Unit (PDU) expands the encoded forms of instructions stored in the Prefetch Buffer and decodes them into a canonical 192-bit internal form that can be executed efficiently by the Execution Unit. Instructions can be encoded in one, three or five 16-bit instruction parcels. These variable length instructions are extracted in two-word blocks from the Prefetch Buffer and entered in an eight-parcel instruction Queue. The instruction Queue aligns

these parcels to fixed decode logic and emits one to five parcels every cycle. The entire PDU can decode and deliver up to two instructions per cycle. Once started, the PDU operates independently, following program text, including branches, fetching instructions from the Prefetch Buffer and depositing 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 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. Branch folding, described below, allows each decoded instruction to represent two instructions from program memory. Every cycle the Decoded Instruction Cache can receive an instruction from the PDU, as well as deliver an instruction 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.[2] It consists of three pipeline stages, with a straightforward sequence of operand fetch, ALU operation, and register writeback. Because CRISP is a memory-to-memory architecture, the EU can calculate addresses, fetch data, 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. Branch folding allows the peak instruction issue rate to be as high as two instructions per cycle. The 192-bit decoded instruction resembles horizontal microcode, and like a 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 microinstruction word is not limited to 32-bits, and unlike CISC machines CRISP does not require a large microprogram ROM. CRISP takes instructions that are compact and easy for a compiler to generate and dynamically transforms them into easy to execute decoded instructions.
- **Stack Cache.** The Stack Cache provides on-chip data storage for the CRISP Microprocessor. The Stack Cache is implemented with two 32-word byte-addressable register files. Every cycle, the Stack Cache provides two independent reads and a single write.

Separate Instruction Fetch and Execution Units

CRISP contains two independent machines on the same chip, the PDU and the EU. These two machines are separated by a Decoded Instruction Cache. In addition to buffering short loops for the Execution Unit, the Decoded Instruction Cache also isolates the PDU and the EU from each other. As long as the EU gets the instructions it needs from the Decoded Instruction Cache, it doesn't bother the PDU. Similarly, the PDU will decode instructions, following the most likely path of their execution, and insert them in the Decoded Instruction Cache.

The PDU typically decodes a few instructions ahead of the EU's needs. By writing an instruction in the cache just before the EU reads it, the apparent hit rate of the Decoded Instruction Cache is much larger than one would expect for a direct-mapped cache of so few entries. Even in cases where the Decoded Instruction Cache has a miss, the PDU often has already started to fetch and decode the desired instruction.

The Stack Cache

Registers are typically used to hold frequently referenced variables inside a CPU in order to reduce memory traffic and speed up operand accesses. Measurements on the behavior of programs have shown that a large portion of memory accesses, approximately 80% for typical programs,[3] are to the stack frame. The stack holds local variables, incoming and outgoing arguments, compiler temporaries and registers

being saved during procedure calls. Measurements also show that these accesses to the stack are on typically only a few tens of words concentrated around the top of the stack. The compiler attempts to move this data into registers whenever possible. Unfortunately aliasing problems prevent most one-pass C compilers from keeping values in registers for more than a single statement unless the explicit keyword **register** is used by the programmer. The result is a substantial amount of memory traffic between a small number of general purpose registers and a few locations on the stack. Figure 2 graphically shows the shuffling of data between registers and the memory based stack.

The CRISP Microprocessor allocates data registers in a way which is radically different from traditional machines. Rather than have the compiler allocate registers and generate code to move data back and forth between registers and the stack, CRISP automatically maps the stack onto machine registers, called the Stack Cache. General purpose registers are eliminated. By tracking the top of the stack in high speed machine registers, useless traffic to and from the stack is avoided and a high degree of register allocation is achieved. Registers are allocated by the hardware, rather than by a software compiler. The Stack Cache automatically maintains the locations near the top of the stack frame in high speed registers. The Stack Cache is a hybrid gaining the best of both caches and registers.

The Stack Cache is implemented as a circular buffer of registers, maintained in a traditional manner with head and tail pointers, called the Stack Pointer (**SP**) and Maximum Stack Pointer (**MSP**). In the CRISP Microprocessor, the SP and MSP are 28-bit registers holding quadword aligned addresses. The MSP is used to delimit the highest address of data which is currently kept in the Stack Cache registers; the SP delimits the lowest address of data in the Stack Cache. If an operand address is greater than or equal to the SP and less than the MSP, then the operand is fetched from the Stack Cache. If if the operand is determined to not be be in the Stack Cache, then it will be fetched from off-chip memory. The Stack Cache registers are built from a pair of 32-entry, 32-bit wide random access memories. Duplicate register sets allow access of two operands simultaneously.

Four instructions are typically used in invoking and returning from a procedure: **call, enter, return** and **catch**. Automatic maintenance of the Stack Cache is done by the **enter** and **catch** instructions. The **call** instruction first moves the return address onto the stack and branches to the target address. The target of a **call** instruction is an **enter** instruction. Enter decrements the SP by the value of its operand to allocate a stack frame for the new procedure. If enough free space exists in the circular buffer for the entire new stack frame, then the **enter** instruction is finished. If not enough free space exists for the new stack frame, some entries nearest the MSP must be flushed back to main memory. If the new stack frame is less than or equal to the size of the Stack Cache, then only the new frame size minus the number of free entries needs to be flushed. If the new stack frame is greater than the size of the Stack Cache, then the entire Stack Cache is flushed and only the part of the frame nearest the SP is kept in the Stack Cache.

The **return** instruction deallocates the space for the current stack frame by adding its operand to the SP, then branches to the return value previously placed on the stack. When control has returned to the calling procedure entries that had been flushed to main memory may need to be restored to the Stack Cache. This is the job of the **catch** instruction. The argument of the **catch** instruction specifies the number of Stack Cache entries that must be valid before the flow of execution can resume. If entries need to be restored, the chip automatically retrieves them from main memory. In a large number of cases no entries will need to be restored, and the **catch** instruction does no work.

The Stack Cache provides the best of both registers and caches. Like registers, access is fast, and no cache tag comparison needs to be done after the data access to determine whether the word is valid. Careful choice of instruction-encoding allows a small offset field to give good code density in the same manner as using a small number of bits for a register index field. In terms of register allocation, since quite often

all local variables will be resident in the Stack Cache, no locally optimizing compiler could possibly have allocated registers better. Unlike registers, however, the Stack Cache can hold strings, arrays and structures. Moreover, one can take the address of a variable in the Stack Cache. The Stack Cache also has the benefits of traditional cache memories in terms of software transparency. Two different machines may have a different number of Stack Cache registers without the compiler needing to be aware of the differences.

Branch Folding

Branches are among the most frequently executed instructions in general purpose computers. For multiple address machines such as CRISP, up to a third of all instructions may be branches. Typically, one third of branches are unconditional; the remainder are conditional. Branches are a problem in high performance machines because they break the smooth flow of instructions though a pipeline, causing the average throughput rate to be much lower than the peak rate.

‘‘Delayed Branch’’ instructions are employed by many RISC machines to alleviate pipeline breakage. While this may make the branch fit into the pipeline better, the delayed branch technique may also require the execution of an extra no-op instruction or two following the branch. In many cases these no-op slots may be filled with useful instructions on one-address load/store machines because their poor instruction efficiency results in larger basic blocks. The delay slots are much harder to fill with useful instructions in a more instruction efficient memory-to-memory style architecture such as CRISP, where the basic block size may average as few as three instructions. Rather than executing branches in a traditional manner or executing more instructions with a delayed branch style, CRISP uses a new technique called *Branch Folding* that can eliminate branch instructions entirely. Branches effectively execute in zero time.

An instruction may be executed several times from the Decoded Instruction Cache and each time (except for conditional branches) the next instruction address is always the same. Instead of recalculating this address every time the instruction is executed, the next address logic is moved to the input side of the decoded instruction cache and a 31-bit ‘‘next address’’ field is added to the cache. As instructions are decoded and placed in the cache, their ‘‘next address’’ value is stored with them. When an instruction is read from the instruction cache, the next address is immediately available to retrieve the next instruction from the cache.

This architecture is similar to many high speed microprogrammed machines where each microinstruction contains a next address field. We have extended this technique to achieve the same benefits for macroinstructions by dynamically generating the contents of the next address field rather than storing it with each instruction in main memory.

Providing a next address field for every instruction in the decoded instruction cache has the same effect as turning every instruction into a branch. Since every instruction in the cache can specify a branch address, there is no need for separate branch instructions in the internal machine. Logic in the PDU recognizes when a non-branching instruction is followed by a branch instruction and ‘‘folds’’ the two instructions together. This single instruction is then placed into the Decoded Instruction Cache. The separate branch instruction disappears entirely from the execution pipeline and the program behaves as if the branch were executed in zero time.

A second address field called the ‘‘alternate address’’ is added to the Decoded Instruction Cache to handle conditional branch instructions. This field is used to hold the second possible next instruction address for folded conditional branch instruction. When an instruction folded with a conditional branch instruction is read from the instruction cache, one of the two paths for the branch is selected for the next instruction address, and the address that was not used is retained with the instruction as it proceeds down

the execution pipeline. The alternate address field is retained with each pipeline stage only until the logic can determine whether the selected branch path was correct or not. When the outcome of the branch condition is known, if the wrong next address was selected, any instructions in progress in the pipeline following the conditional branch are flushed and the alternate address from the folded conditional branch is re-introduced as the next instruction address at the beginning of the execution pipeline.

Determining which of the two possible next addresses of a conditional branch is likely to be taken is aided in CRISP with static branch prediction. The encoding for the conditional branch instruction contains a single branch prediction bit which may be set by the compiler. From measurements on C programs, we have found that a single static branch prediction bit can yield results within a few percent of more complex dynamic prediction strategies. If the branch prediction bit is set optimally, branches can be predicted correctly between 70 and 95 percent of the time.

Branch prediction is useful when a conditional branch instruction in the pipeline can alter the flow of instructions before the result of a comparison can be computed. If, however, there are no compare instructions in the pipeline then there is no need for branch prediction. Since only the compare instruction may set the condition code flag, the outcome of the conditional branch is known with certainty. As a conditional branch instruction is latched into the first pipeline stage of the Execution Unit the already determined outcome for the branch can be used to select the correct next PC address. Without intervening compare instructions in the pipeline an effective 100% correct prediction rate can be achieved.

CRISP intentionally has separate compare and conditional branch instructions so that a compiler or optimizer may insert instructions between the compare and conditional branch. This form of code motion, for which our terminology is *Branch Spreading*, is similar to that used with delayed branch instructions.[4] Compared to the delayed branch scheme, the use of a delayed branch instruction still costs the delay of a full instruction to move the branch through the pipeline. By combining branch folding and code motion in CRISP there can be no cost at all for either conditional or unconditional branches.

Branch Folding and associated hardware/software techniques eliminate many of the problems with branches that have plagued high performance computer designs. The number of instructions issued to the pipeline to execute a given program can be reduced by the number of branches in the program. This can be as much as 30% of the instructions executed. The resulting reduction in the number of instructions executed (and elimination of pipeline breakage) assist CRISP to achieve high performance.

Fast Procedure Calls

In addition to their other uses, the Stack Cache and Decoded Instruction Cache contribute to CRISP's fast procedure calls. The **call** instruction functions like a move combined with a branch. It moves the PC into the address pointed to by the SP. CRISP uses the convention that the first word on the stack is always available to store the return PC. The next address field of the **call** instruction is the starting address of the procedure. The **call** instruction executes in 1 clock cycle.

The **enter** instruction functions like a load address with the SP as the destination. In effect, the operand of the **enter** instruction is subtracted from the Stack Pointer. The **enter** instruction may need to flush entries in the Stack Cache and adjust the MSP. Most of the time however, no entries will need to be flushed, and the operation will complete in 1 clock cycle.

The **return** instruction is slightly more complicated. **Return** must first add its argument to the SP, requiring 1 clock cycle, before it can move the return PC (as now pointed to by the new SP) back into the Program Counter to effect the return branch. **Return** executes in 2 clock cycles.

Catch, like **enter**, will most often have no entries to restore to the Stack Cache. In this case **catch** functions as a no-op, and takes 1 cycle.

Only five clock cycles are required to complete the calling sequence – calling a procedure, allocating space for a new stack frame, de-allocating that space, returning, and then executing a **catch** instruction.

Passing arguments may also be considered part of procedure call overhead. CRISP intentionally avoids ''push'' and ''pop'' instructions that modify the SP and interfere with pipelined implementations. Instead, by software convention, space is pre-allocated for all outgoing arguments. Arguments are placed on the stack with simple move instructions or computed in place. Since many procedures have only one argument, we have made the Accumulator coincident with the same stack location as the first outgoing argument.

Read Cancelling

A *hazard* exists in a pipelined computer when the address of an operand from one instruction is the same as the destination of another preceding instruction still in the pipeline. Such hazards must be resolved for correct program operation. Hazards can exist either with internal registers or external memory data. Several mechanisms can be used to resolve the hazard. Many pipelined machines recognize the conflict and *stall* an instruction at the operand fetch stage until all instructions that could alter its operands have cleared the pipeline. A good optimizing compiler can recognize this effect and attempt to put other instructions between the store and the fetch of the data, thus avoiding any stalling. Other pipelined machines don't bother to recognize the hazard at all, so the compiler must insert no-ops or move instructions apart in order to insure correct program operation.

The CRISP Microprocessor takes a different approach. Hazards are detected between the store stage of the pipeline and the operand fetch stage, whether the operands are in memory or located in the on-chip Stack Cache. Bypass data busses run back from both the ALU stage and the store stage of the EU pipeline. When a hazard is detected the operand fetch is suppressed and the data comes directly from the instruction that is about to store at the hazard address. Instructions in the operand fetch stage do not have to wait for stores to complete, and the compiler is not required to move instructions around or insert no-ops in order to insure correct operation.

For operands which are found in the Stack Cache, suppressing the operand fetch because of a bypass does not improve the time it takes for an instruction to go from the operand fetch to the ALU stage. It takes one cycle to fetch data from the Stack Cache and one cycle to route the data. However, for operands which are located in external memory, the memory fetch is suppressed, and the instruction proceeds down the pipeline as quickly as if its operands were located in the on-chip Stack Cache. This saves execution time and reduces off-chip bus traffic. We refer to this technique as *Read Cancelling*. An optimizing compiler for the CRISP Microprocessor attempts to move instructions which use the same data closer together, rather than spreading them apart.[5]

A Memory Intensive Design

It is very difficult to continue designing new microprocessors to use the higher number of transistors provided by each new advance in technology. However, it is relatively easy to increase memory sizes as IC technology improves, since memory structures are so regular. The CRISP Microprocessor exploits this technological fact with extensive use of internal caching. The PDU and the EU are capable of issuing instructions at approximately the same rate, but only if they are kept supplied with program text and decoded instructions, respectively. In the current implementation, the performance of CRISP Microprocessor is limited by I/O bandwidth, so the PDU cannot always deliver instructions at the rate the EU can

complete them. This can cause the average instruction issue rate to fall below one instruction per cycle. However, with improving VLSI technology the sizes of the Prefetch Buffer and the Decoded Instruction Cache can be increased, with little architectural redesign. Larger caches will improve performance since off-chip I/O traffic is reduced and instructions are more likely to be cache resident and available for immediate execution.

The instruction sets of early computer architectures reflect the high cost of register memory. Some early machines had only one register and their instruction sets were designed around this fact. By the 1960's, it was economical to build 8 to 16 registers, so the instruction-sets had three or four bit register fields. Now 32 registers are considered reasonable, but the older architectures are locked into a smaller number by their instruction encoding. Explicit registers are not present in the CRISP instruction set, so future implementations can freely increase the size of the Stack Cache for improved performance without changing the instruction encoding.

Conclusion

CRISP is a very unusual microprocessor. Branches can be executed in zero time. Registers are allocated dynamically by the hardware rather than by a compiler. Autonomous execute and decode units work cooperatively to execute a common program. The overhead for procedure calls has been reduced dramatically from that of typical register oriented machines. All these features are available on a machine with an instruction set that is simple to understand and for which it is easy to generate code. CRISP achieves the instruction efficiency and code compaction of CISC machines, while reaching beyond the performance levels of pure RISC machines without resorting to extremes of compiler optimization.

References

1. A. Berenbaum, D. Ditzel, H. McLellan, *An Introduction to the CRISP Architecture,* Proceedings of the Spring 1987 COMPCON.
2. 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).
3. D. R. Ditzel, 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).
4. J. L. Hennessy, T. R. Gross, ''Optimizing Branch Delays,'' Computer Systems Lab Technical Report, Stanford University (1981).
5. S. Bandyopadhyay, V. S. Begwani, R. B. Murray, ''Compiling for the CRISP Microprocessor,'' *Proceedings of the Spring 1987 COMPCON*.

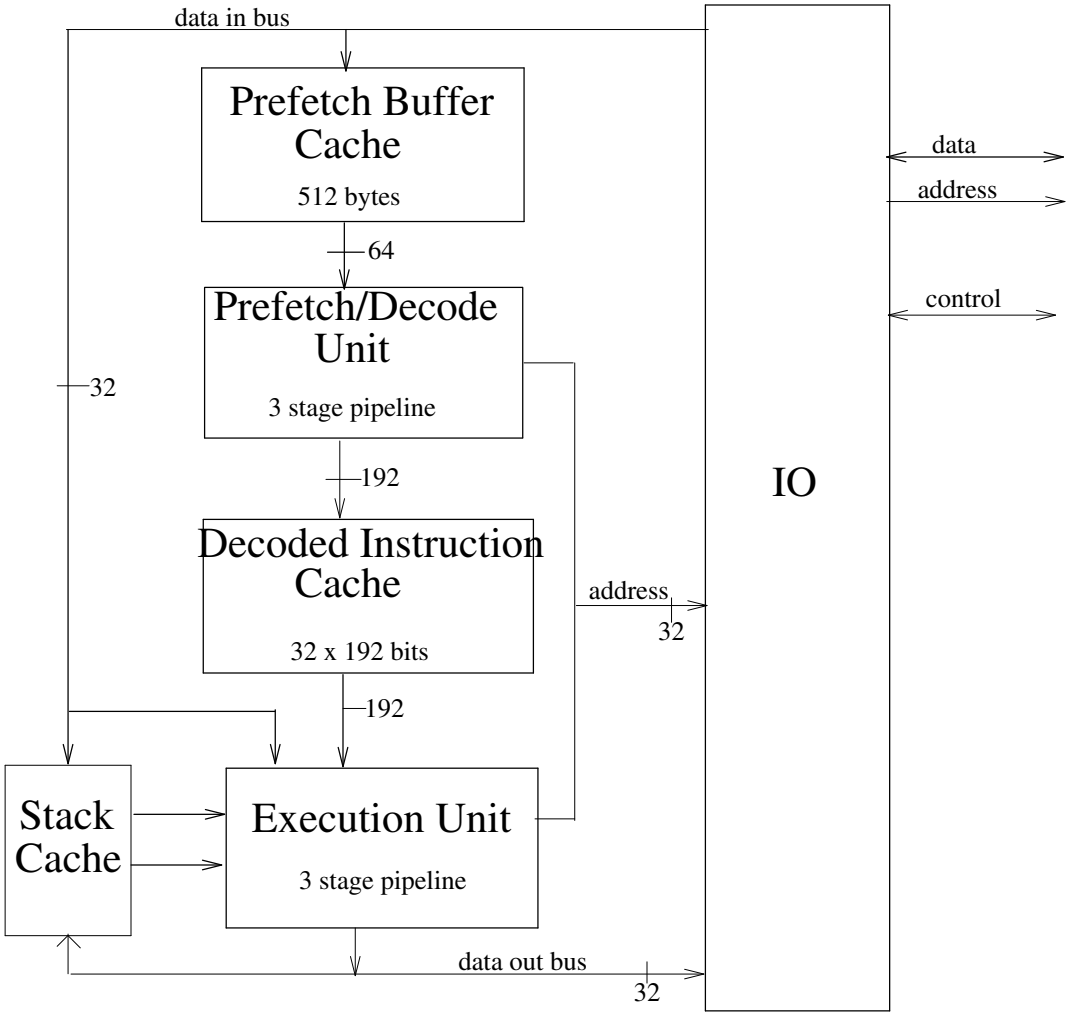


Figure 1. CRISP Microprocessor Block Diagram

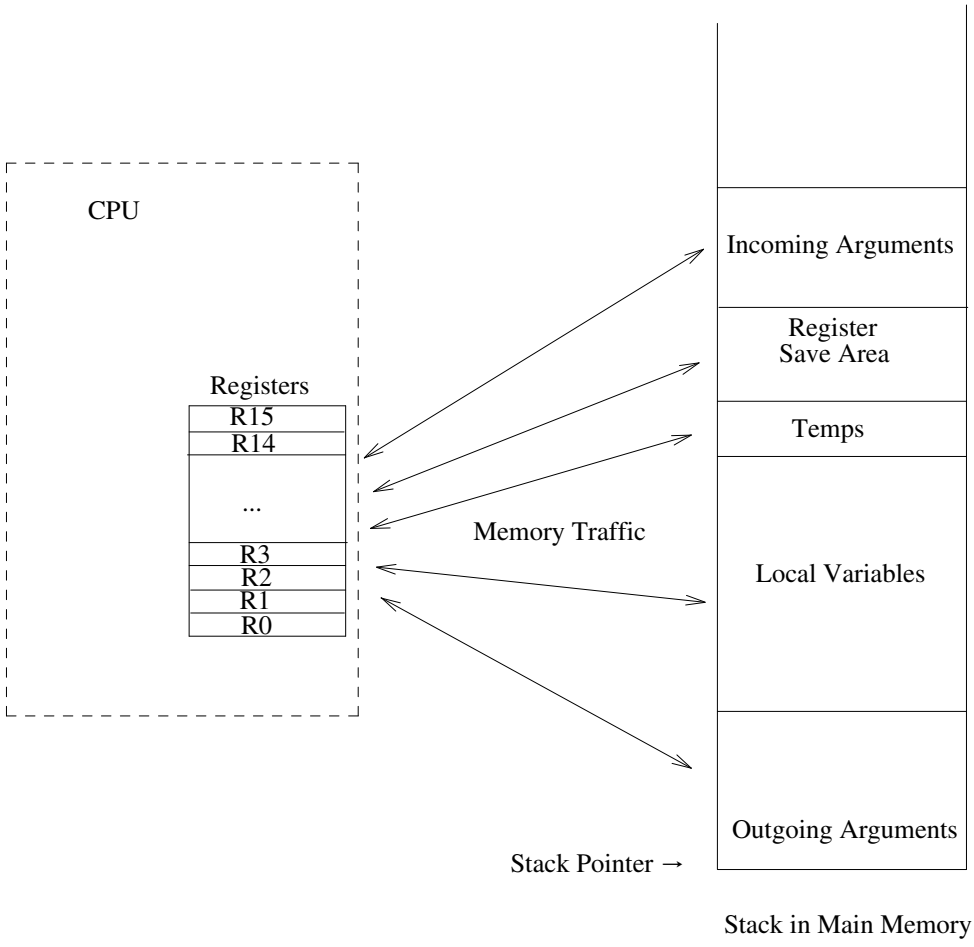


Figure 2. Traditional Memory Traffic Between Registers and the Stack.