A Big RISC

Richard A. Blomseth, Capt., USAF

Masters Project Final Report

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Big RISC (BRISC) is a high-speed CPU designed with 100K ECL logic and based on the RISC I architecture. Design, performance, and cost of BRISC is presented. Performance is shown to be better than high end mainframes such as the IBM 3081 and Amdahl 470V/8 on integer benchmarks written in C, Pascal and LISP. The cost, conservatively estimated to be $132,400, is about the same as a high end minicomputer such as the VAX-11/780. BRISC has a CPU cycle time of 46 ns, providing a RISC I instruction execution rate of greater than 15 MIPs.

BRISC is designed with a Structured Computer Aided Logic Design System (SCALD) by Valid Logic Systems. An evaluation of the utility of SCALD for computer design is also included.

July 18, 1983

The work reported herein was supported in part by Defense Advance Research Projects Agency (DoD) ARPA Order No. 3803, Monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251, and the U.S. Department of Energy under Contract DE-AT03-76SF00034, Project Agreement DE-AS03-79ER10358.

Richard A. Blomseth
Author

A Big RISC

Title

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, to partial satisfaction of the requirements for the degree of Master of Sciences, Plan II.

Approval for the Report and Comprehensive Examination:

Committee: D. A. Patterson, Research Adviser

July 18, 1983 Date

John L. Hennessy, Second Reader

July 17, 1983, Date

Table of Contents

A Big RISC

Richard A. Blomseth, Capt., USAF

Masters Project Final Report

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

1. Introduction

Even though microprocessors are now available with the power of previous years' minicomputers, and are expected to get faster, there will always be problems where even the fastest microprocessors are not fast enough. This paper describes the design of BRISC (Big Reduced Instruction Set Computer), a high performance 32-bit processor designed with discrete 100K ECL logic. BRISC uses the same concepts used to speed up the RISC I microprocessor; [14] these concepts include a simplified instruction set and overlapping register windows.

BRISC is designed with a Structured Computer-Aided Logic Design (SCALD) system. [11] Most of the CPU design has been entered into the SCALD system and post processed for physical design errors. The worst case logic path has been timing verified using SCALD to approximate the CPU cycle time. Some simulation has been done (about 70% of the design) to verify functional correctness of the CPU design.

This paper describes the design, performance and cost of BRISC. Performance and cost are compared to other high performance computers to evaluate the usefulness of the RISC architecture for high performance computer design. The rest of this section describes the RISC I architecture and the Berkeley RISC implementations. RISC I architecture is described because BRISC uses the RISC I architecture at the instruction set level.

1.1. RISC I Architecture

Fundamental to RISC architectures is the concept that frequent, time consuming tasks should be done as fast as possible, and infrequent tasks may be done more slowly. RISC I is designed to execute high level languages such as C and Pascal efficiently, so the RISC I instruction set concentrates on the most time consuming operations of high level languages. Two time consuming operations optimized by the RISC I instruction set are: (1) procedure calls and returns, that account for 40% of the time spent in traditional architectures; and (2), data references to local scalar variables and constants, that account for over 60% of data references. [14] RISC I is

designed for integer programs; floating point operations are done in software and are therefore significantly slower than integer operations.

The remainder of this section provides a brief introduction to the RISC I architecture. For more information, the reader is referred to the *RISC I Principles of Operation.* [25]

RISC I has two significant departures from traditional computer architectures. First is a small and simple instruction set; second is a large register file consisting of overlapping register windows.

1.1.1. RISC I Instruction Set

The RISC I instruction set consists of 31 instructions and supports five addressing modes. Three address modes – register, immediate and indexed – are supported directly by the instruction set. Two addressing modes – absolute and register indirect – may be synthesized from indexed addressing. Only the *LOAD* and *STORE* instructions access the main memory, all other instructions read and write registers. Most operations read two registers and write into a third. Because of the register-to-register orientation of the instruction set, RISC processors complete an instruction nearly every memory cycle. *LOAD* and *STORE* instructions are the only exceptions and require one or two extra cycles.

RISC/E added 23 instructions to the RISC I instruction set to support relative loads and stores, increased access to the special registers, and support for the virtual memory. BRISC uses the RISC/E instruction set. The BRISC instruction set is included in Appendix A.

1.1.2. Register File

RISC I relies on multiple sets or *windows* of 32-bit registers to speed up calls, returns, and access to local variables. This collection of register windows is known as the *register file.*

BRISC contains a register file consisting of 128 32-bit registers. Sixteen of these registers (the register window) are available to a procedure at any one time, and a new set is allocated for a *CALL* and deallocated for a *RETURN.* Memory accesses are not required to save and restore the return address or other registers as required by conventional architectures. Furthermore, movement of data is not required for parameter passing because allocation and deallocation of registers for *CALL* and *RETURN* is done with overlapping windows. Overlapping register windows usually allow the program's complete activation stack to remain in registers, so that the CPU references fast registers, rather than slow main memory.

BRISC has two sets of 128 registers, one set is active in user mode, the other is active in system mode. RISC I registers *spill* into memory if the 128 register limit is exceeded. Ten of the 16 registers in a register window are shared by all windows in a register file and are known as the

global registers.

1.2. Berkeley RISC Implementations

Four RISC processors have been designed at UC Berkeley since the project was started in 1980. Two of them are single chip NMOS designs (RISCs I and II), and two are discrete logic ECL designs (RISC/E and BRISC).

RISC I

RISC I is the first RISC designed at Berkeley. It was designed by five students in 6 months. The 44,500 transistor chip was fabricated using 4 micron NMOS technology, and is 10.3 x 7.74 mm (406 x 305 mils). Through the use of computer aided design tools, RISC I worked on first silicon. Even though the chips work, they do not meet their design goal for speed. The design goal for RISC I was a 400 ns cycle; RISC I runs with a 2000 ns cycle. Even at 2000 ns RISC I runs C programs faster than an 8 MHz 68000. [14]

RISC II

RISC II was designed by two graduate students in two years and is a more refined processor than RISC I. Building on the success of RISC I in using computer aided design tools to design chips that were logically correct, RISC II used a newer set of tools that included a program to verify the speed of the chip (Crystal). [10] The longer design cycle and better tools resulted in a better circuit design and working chips with 75% more registers (138), 25% less area (10.3 x 5.8 mm), and four times the speed (500 ns cycle time) of RISC I.

RISC II uses a three stage pipeline as opposed to the two stage pipeline of RISC I. A similar three stage pipeline was subsequently used by RISC/E and BRISC. The three stage pipeline is discussed in Section 2.1.

RISC/E

RISC/Extended (RISC/E) was a paper design of a 10K ECL CPU and cache also started in 1980 to see if the RISC I concepts could be applied to a high performance discrete logic CPU. [5] The result is the design for a fast CPU with a 75 ns cycle time consisting of 450 10K ECL parts. The cache also has a cycle time of 75 ns and consists of 550 10K ECL parts.

BRISC

Big RISC (BRISC) continues the work started by RISC/E to verify the applicability of RISC concepts to high performance computer design. Jeff Deutsch and the author designed the first version of BRISC during the Winter quarter of 1983. BRISC started as a translation of the 10K ECL design of RISC/E to 100K ECL. Minor changes were made throughout the design to accomodate differences between the 10K and 100K parts. The ALU, shifter, program counters, and control logic were redesigned for increased speed and decreased parts count. Timing was kept nearly identical to RISC/E.

The resulting design from the Winter quarter consisted of 606 parts. The high part count was due primarily to the lack of buffers in the SCALD library to prevent loading violations; instead parts were replicated to obtain the desired drive. Simulation was completed on the ALU, shifter and program counters, and post processing was started. Timing verification was not performed because of bugs in the timing verifier.

The author continued work on BRISC during the Spring quarter of 1983. BRISC was redesigned to use buffers that were added to the SCALD library, and control was redesigned to require fewer bits. The resulting design uses 332 parts. The BRISC equivalent of microcode was written for the arithmetic instructions, and simulation was started for the entire design. The timing verifier became available from Valid, so enough of the design was verified to calculate the cycle time. Post processing was completed for the entire design.

BRISC is faster than RISC/E (47 ns vs. 75 ns) with fewer chips (332 vs. 550). 100K ECL is a faster logic family than 10K ECL, but was not widely available when the RISC/E project was started (1980). The 100K ECL parts tend to have more functionality than 10K ECL parts, but this functionality is bought at the expense of more pins per package. Nearly all 100K ECL parts have 24 pins, whereas most 10K ECL parts have 16 pins.

The remainder of this paper describes the design, performance and cost of BRISC.

2. BRISC Design

BRISC's design is based on the three stage pipeline developed by Lloyd Dickman that was used by RISC II and RISC/E. This section describes the pipeline timing and the hardware design of BRISC.

2.1. Pipeline Timing

BRISC pipeline timing will only be summarized here, a more complete description may be found in the RISC/E design study. [5] BRISC uses a two phase, non-overlapping clock. Most instructions take three cycles, or six phases, to execute. The CPU is pipelined so that three

instructions are being acted on simultaneously, allowing instructions to issue and complete at a rate approaching one per cycle. Figure 1 describes the six phases of a typical instruction, and Figure 2 shows three instructions passing through the pipeline. The pipeline is designed to make maximum use of key resources – the cache, register file, and ALU – and to minimize contention for these resources. The pipeline is also designed to not require pipeline flushes, all instructions entering the pipeline go through to completion unless externally interrupted.

Most instructions follow the pipeline timing shown in Figures 1 and 2 so that no complex pipeline interlocks are required. Even instructions that receive special treatment in traditional pipelines - such as *CALL*, *JUMP*, and *RETURN* - do not affect pipeline timing in BRISC and follow the same basic three cycle timing.

Only the *LOAD* and *STORE* instructions require more than three cycles to execute. One extra cycle is required for the memory access; a second extra cycle is required for 8- and 16-bit loads and stores to align the memory data with the register file using the shifter. The second extra cycle is not required for *LOAD WORD* and *STORE WORD* because 32-bit words are always aligned with memory.

2.2. BRISC Hardware Design

Figure 3 is the top level SCALD drawing for BRISC (Appendix E contains the complete set of SCALD drawings for BRISC). The BRISC CPU contains the ten modules shown in Figure 3. The control logic is in the *CONTROL* module and the other nine modules constitute the data path. Design of the BRISC data path and control is described in the following sections.

2.2.1. Data Path

Most data flows from left to right in Figure 3, starting with the instruction address being generated in *PCs AND ADDRESS* and ending with the result in the *RESULT LATCH* for subsequent writing back to the *REGISTER FILE*. The following paragraphs describe each of the modules in the data path.

2.2.1.1. PCs AND ADDRESS

The top half of the *PCs AND ADDRESS* drawing in Appendix E contains the program counters (PCs). BRISC has three PCs: the *Next PC* (PCN), *Current PC* (PCC), and *Last PC* (PCL). The PCN is incremented at the start of the instruction cycle, used to fetch the next instruction, and, except for jumps, becomes the PCC at the end of the instruction cycle. The PCC holds the value of the program counter for the currently executing instruction and the PCL holds the 'last' value of the PC, useful for interrupts and exceptions.

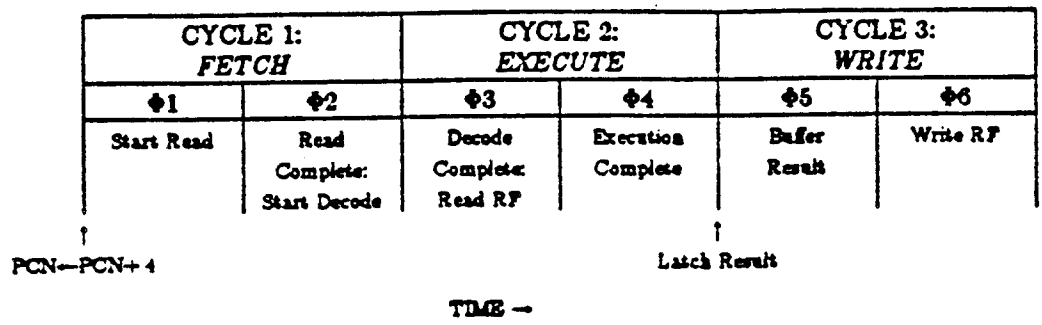
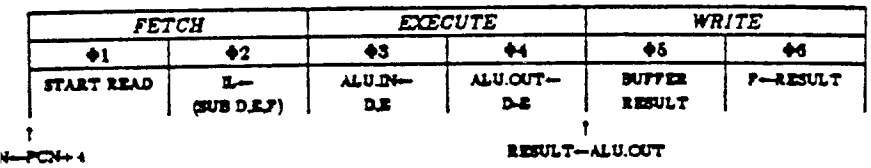
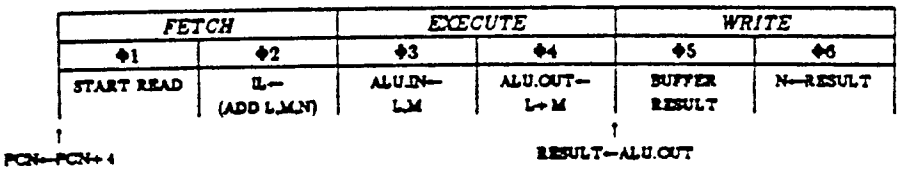


FIGURE 1. BASIC TIMING CYCLE. The three cycles of an instruction, and correspondingly the three stages of the execution pipe, are called *Fetch*, *Execute*, and *Write*. At the start of phase 1, the Next Program Counter (PCN) is gated onto the cache address bus and by the end of phase 2 the instruction has been decoded. In phase 3, the two operand registers are read from the register file and the Arithmetic Logic Unit (ALU) and shifter operations are selected. By the end of phase 4, the ALU and shift operations are complete. In phase 5, the result of the data-path operation is buffered and the sign optionally extended. Finally, in phase 6 the result is written back into the register file.





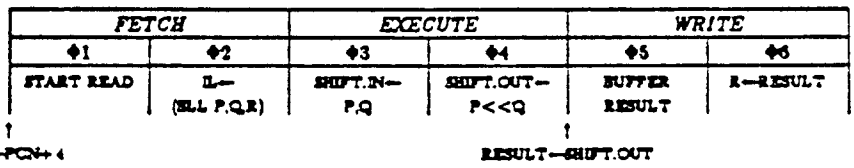


FIGURE 2. STANDARD PIPELINE TIMING. The following instruction stream is shown traversing the pipeline:

SUB D,E,F
ADD L,M,N
SLL P,Q,R

The x-axis is time. A vertical line drawn through the three timelines would show up to three simultaneous operations occuring in the CPU.

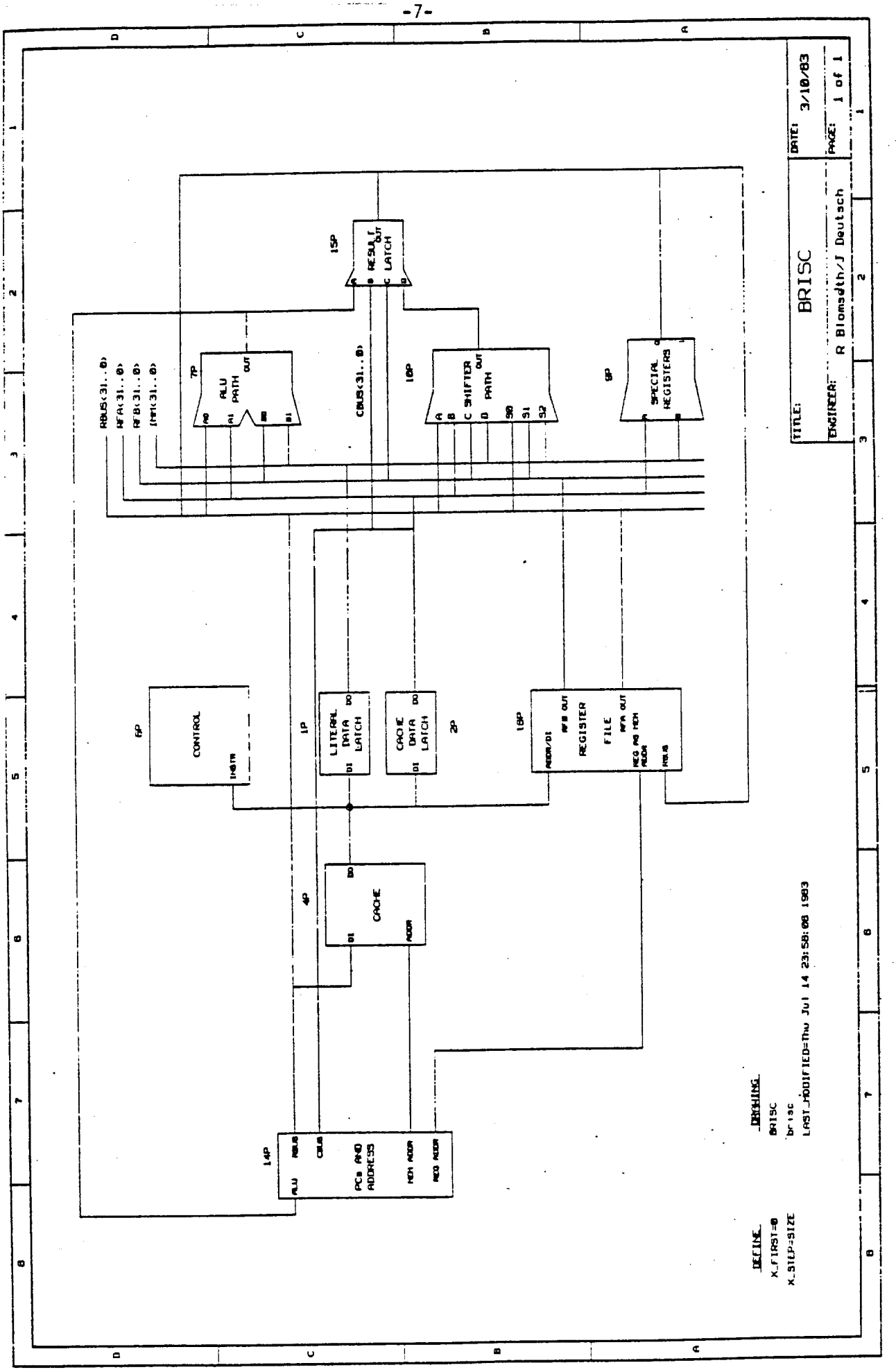


Figure 3. BRISC Top Level Diagram

The bottom half of the *PCs AND ADDRESS* drawing shows the memory address generation. There are two sources for the memory address going to the cache, one is the PCN for instruction fetches, the other is the ALU for loads and stores. The Cache Address Multiplexor selects the correct address source and sends it to the *CACHE* module.

2.2.1.2. CACHE

A cache has not been designed for BRISC. The 'cache' in the drawings is a 256 by 32-bit memory used only to enable SCALD simulation of the BRISC design. The actual cache would be on a separate board and would be similar to the cache designed for RISC/E. The RISC/E cache board contains a cache, Translation Buffer, and Page Usage Buffer.

RISC/E Cache

The RISC/E cache is a 2-way set associative data and instruction cache with 2048 32-bit words. The line size may be set by software and is variable from 4 to 32 words. The number of sets is variable from 64 to 1024. A write-back policy and least recently used (LRU) block replacement algorithm is used.

RISC/E Translation Buffer (TLB)

The purpose of the TLB is to store the most recent virtual to physical memory address translations. Accesses to virtual addresses not in the TLB are trapped and translated by software. The TLB is a 2-way set associative buffer that stores 1024 address translations and associated memory protection and replacement information.

RISC/E Page Usage Buffer (PUB)

The PUB contains reference and modification history for each physical page frame. This information is updated by hardware and used by the page replacement software to identify pages to be replaced when more physical page frames are required.

2.2.1.3. REGISTER FILE

The *REGISTER FILE* module contains two sets of 128 32-bit registers used for the system and user register files. Two independent read requests, or a single write request, may be serviced in a single clock phase. The dual read is implemented by using two redundant copies of the registers. Both copies are written simultaneously, but are read independently to retrieve two operands at a time. The register file memory is implemented with eight 256x4-bit Fujitsu ECL RAMs. The RAMs have a 7 ns access time.

The *REGISTER FILE* module supports mapping the 128 registers into the virtual address space so that executing processes may access any of the 128 registers by referencing the appropriate virtual address.

2.2.1.4. ALU PATH

The *ALU PATH* module contains a high-speed ALU and carry lookahead unit, input multiplexors and input/output latches. Each of the two inputs of the ALU can accept data from two independent sources. The sources may be either of the two source registers specified in the instruction, immediate data contained in the instruction, the Current PC for relative addressing, or the result of the previous instruction. *ALU PATH* output is routed to the *RESULT LATCH* for writing to the *REGISTER FILE* and also to the Cache Address Multiplexor in *PCs AND ADDRESS* to be used as the memory address for branches, loads and stores. All ALU operations execute in a single clock phase.

2.2.1.5. SHIFTER PATH

The *SHIFTER PATH* module performs 1- to 32-bit left or right shifts with optional sign extension. Input multiplexors and latches select from the same set of inputs used by the *ALU PATH*. The output is latched and routed to the *RESULT LATCH* for writing to the *REGISTER FILE*.

The basic shift part provided in 100K ECL can only do a logical shift right or a rotate. The BRISC instruction set also requires arithmetic shifts and left shifts. Arithmetic shifting is provided by extending the sign bit of the 32-bit operand to an additional 32 bits and making it the upper 32 bits of the 64-bit input to the shifter. Left shifting is provided by having a path into the shifter that reverses the input bits and then reverses the result bits on output.

2.2.1.6. SPECIAL REGISTERS

The *SPECIAL REGISTERS* module contains the Program Status Word (PSW), Current Window Pointer (CWP), and Saved Window Pointer (SWP). The PSW holds the ALU condition codes and a byte of CPU status flags. The CWP contains the counters and registers used to point into the currently active position in the system and user register files. The SWP points to the last words of the system and user register files that are in memory and provides detection of register window overflow and underflow for *CALL* and *RETURN*.

2.2.1.7. RESULT LATCH

The *RESULT LATCH* module is composed of a 4-way multiplexer and a 32-bit latch. It serves to buffer the result of a computation at the end of each cycle. The four inputs to the *RESULT LATCH* are the *ALU PATH* output, the *SHIFTER PATH* output, one *REGISTER FILE* output and the Current PC from *PCs AND ADDRESS*. The output is routed to the *REGISTER FILE* for register writes and to the result bus (RBUS) for register forwarding. Register forwarding is required when an instruction requires the result of its preceding instruction. Because of the nature of the pipeline, the result will not have been written back to the register file, so it is routed directly to the input multiplexors.

2.2.2. Control

RISC processors achieve high performance by optimizing the data path, not by expanding control. The simple instruction set results in simpler control than is found in conventional processors. Control consists of pipelined latches driven by control RAMs. The RAMs are used to decode instructions and control the data path. Control is described in terms of hardware, contents of the control RAMs, and the support processor used to program the control RAMs.

2.2.2.1. Control Hardware

The *CONTROL* module consists of a pipeline of three latches that correspond to phases 3, 4 and 5 of instruction execution. Instructions are stepped through the pipe and decoded during phases 3 and 4. Movement of control data through the control pipeline corresponds to the movement of data through the execution pipeline in the data path.

Decoding of instructions is done with RAMs in the *PHASE 3 DECODE RAM* and *PHASE 4 DECODE RAM* modules. Input to the RAMs consists of the opcode and immediate fields of the executing instruction, the outputs from the RAMs are the control signals for the data path.

The pipeline is controlled by another set of latches in the *PIPELINE CONTROL* module. Input to *PIPELINE CONTROL* consists of the *Pipe Control* bits generated by the *PHASE 3 DECODE RAM*. These bits control insertion of one or two extra cycles for *LOAD* and *STORE* instructions.

2.2.2.2. Control 'Microcode'

Stone defines microprogramming as 'the use of storage to implement the control unit.' [1] By this definition, the contents of the decode RAMs in *CONTROL* are the BRISC equivalent of microcode. BRISC microcode, however, is much simpler than traditional microcode. Traditionally, a variable number of microinstructions are executed per machine instruction. Sequencing of

microinstructions is usually controlled by bits in the microinstructions and a microinstruction counter. BRISC always fetches exactly two microinstructions per RISC instruction, thus BRISC does not require a microinstruction counter or complex sequencing control. Sequencing of the microcode is controlled only by the instruction stream; each opcode field is decoded to the 54-bit microcode word during Phases 3 and 4. Only two bits affect timing in any way, they are the *Pipe Control* bits described above.

A microcode assembler called *DAPL* is used to create microcode for BRISC. [15] Only a subset of the full capability of DAPL is required because of the simplicity of the BRISC microcode. DAPL is used primarily to allow for symbolic names instead of absolute bit patterns in order to identify the field names and field contents of the microcode. Symbolic field names and contents simplify shuffling bits as the microcode is finalized.

The microcode for BRISC has not been completed. Enough microcode has been written to allow simulation of the BRISC arithmetic instructions. A listing of the microcode as DAPL input is included in Appendix F.

2.3.2.3. Support Processor

The support processor initializes contents of the control RAMs. The support processor may also be used to debug the BRISC hardware. Serial inputs and outputs of the edge-triggered shift registers used as latches in BRISC may be tied together to allow scan-in scan-out of processor state by the support processor. Some of the flow-through latches currently used by BRISC may be converted to edge-triggered shift register latches to provide more of the processor state to the support processor. Additional hardware may be added to allow single stepping the BRISC CPU with the support processor.

The interface to the support processor has not been designed, nor has a support processor been selected. It is envisioned that the interface to the support processor will have minimal impact on the BRISC design, and that any commercially available micro or minicomputer may be used for the support processor.

3. Performance Analysis

BRISC performance is estimated in terms of benchmark performance, and is compared to the same benchmarks run on other high performance computers.

3.1. BRISC Performance

BRISC performance is determined by the speed of the CPU, memory, and Input/Output (I/O) subsystems - and also by the degree of overlap of each. Degradation of performance by the I/O subsystem is assumed to be minimal for the benchmarks used, and is not considered. The benchmarks do not include system overhead because task time is measured instead of elapsed time; therefore, system overhead is not considered.

As a result of ignoring I/O and system overhead, only CPU and memory subsystem speed is used to determine BRISC performance. CPU speed is first considered to determine the CPU cycle time; then this time is degraded by the memory subsystem overhead to calculate an estimate of BRISC performance.

BRISC performance calculation can only be an approximation because of the number of variables that affect performance. As a result, three BRISC performance figures are calculated using three sets of assumptions for the performance variables. The three performance figures provide a simple sensitivity analysis of the assumptions made, and are identified as BRISC A, BRISC B and BRISC C. BRISC A is a fictitious best case figure, unattainable but a reasonable upper bound on performance. BRISC B is a middle of the road figure, with conservative estimates used to calculate a plausible estimate of BRISC performance. BRISC C uses worst case figures to give a lower bound on performance.

3.1.1. CPU Speed

CPU speed is determined by the basic cycle time of the CPU and the number of cycles per instruction. These numbers are presented, and suggestions made for possibly improving the BRISC CPU speed.

3.1.1.1. CPU Cycle Time

The SCALD system timing verifier was used to determine the BRISC CPU cycle time. The SCALD timing verifier uses worst case minimum and maximum delay times through all parts of the design to find timing errors. Output of the timing verifier may be used to calculate worst case logic delays. The timing verifier identified the register file read as the worst case path through the CPU. The register file is in the critical path because of the time required to do the register address calculation and the time required to read the register file RAM (7 ns access time). The minimum time for a register file read including address calculation and RAM access time is 18.5 ns. Assuming equal length phases, the BRISC minimum cycle time is therefore 37 ns.

A 37 ns cycle time assumes worst case logic delays, but no wire delays. Wire delays account for about half the cycle time of many high performance CPUs, so the 37 ns cycle time for BRISC

needs to be corrected for wire delays. The SCALD system does not include a physical design system, so accurate values for the wire delays in BRISC are not available.

SCALD does provide a means to input an estimate of wire delay. Wire delay input is specified as a minimum and maximum delay to be used on all wires. The minimum and maximum numbers are used to find worst case delays as determined by worst case wire delays and worst case logic delays. Figure 4 shows the affect of different values of worst case wire delays on the BRISC cycle time as reported by the SCALD timing verifier.

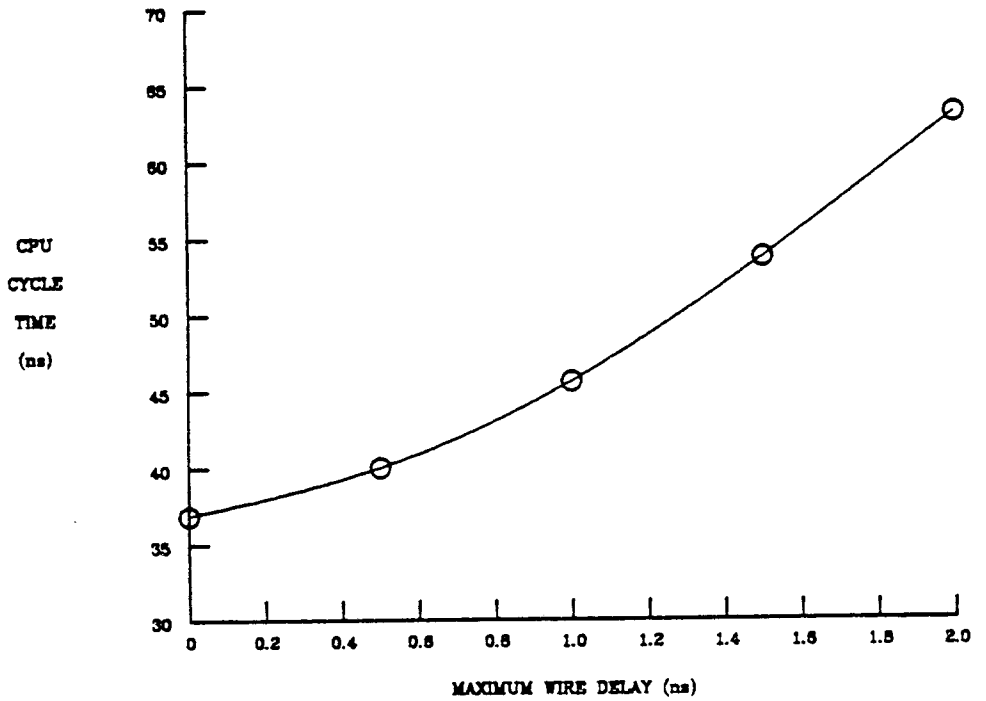


Figure 4. BRISC CPU Cycle Time

The BRISC CPU has been designed to fit on a single board to keep wire lengths short and to avoid the delay penalty of connectors to other boards. It is assumed that chips will be located on the CPU board such that wire delay is minimized through the worst case path. For example, the register file address generation logic will be located in a small area to minimize worst case wire delays.

A best case wire delay of 0.0 ns is used for BRISC A, a median wire delay of 0.5-1.0 ns is used for BRISC B, and a worst case wire delay of 0.5-2.0 ns is used for BRISC C. The resulting CPU cycle times are 37 ns for BRISC A, 46 ns for BRISC B, and 63 ns for BRISC C. A median wire delay of 0.5-1.0 ns is considered reasonable because most of the parts in the critical path can be kept together. Adjacent ICs have a 0.5 ns delay between them, so most wires in the critical

path should be close to a 0.5 ns delay. A few wires may be longer than 2.0 ns, but the average in the worst case path is estimated to be 1.0 ns.

3.1.1.2. CPU Cycles Per Instruction

As stated in Section 3.1, BRISC completes an instruction every cycle except for the *LOAD* and *STORE* instructions that add an extra one or two cycles. RISC I requires one extra cycle for *LOAD* and *STORE*, and this extra cycle was considered in the original RISC I benchmarks. These benchmarks will be used to calculate BRISC performance, so the affect of a single extra cycle for loads and stores will be included. RISC I did not require a third cycle for unaligned loads and stores as required by BRISC. The benchmarks considered, however, only use 32-bit data, so all loads and stores in the benchmarks require only two cycles, and the third cycle may be safely ignored.

The affect of unaligned loads and stores on programs other than the benchmarks should be minimal. Loads and stores can be conservatively estimated to occur in 20% of RISC I instructions. Even if half of the loads and stores are unaligned, the total affect on BRISC performance is an 8% degradation in throughput. This performance degradation may be avoided by using only 32-bit data.

3.1.1.3. Design Changes

An advantage of using SCALD for designing a computer is that experiments may be tried with slightly different architectures to find the overall impact on performance and cost of alternative architectures. An architectural feature is justified only if the benefits of the feature outweigh the cost. For example, if a feature adds 3 ns per phase (6 ns per cycle) to the critical path because of a single logic delay and a wire delay, then that feature increases the 47 ns cycle time of BRISC B by 13%. The feature is therefore not worthwhile unless it makes up for the 13% decrease in CPU cycle time.

Even if a feature does not add gate delays to the critical path, if additional chips are required, the room required by the additional chips may add wire delays to the critical path, and the additional chips add to the total cost. Performance and cost impact is much more severe if an additional board is required.

A few architectural experiments have been run on the BRISC design. The *ALU PATH, SHIFTER PATH,* and *CONTROL* modules were redesigned several times to decrease parts count and increase speed. This section describes some further experiments that should be run to find the cost of specific architectural features. The changes are grouped under design deletions, design enhancements, and other changes.

Design Deletions

SEPARATE SYSTEM AND USER REGISTER FILES: The RISC I architecture does not include separate system and user register files. The separate files were added by RISC/E because the single 128-word register file required by the RISC I architecture occupied only half of the 256-word RAM. The unused half of the RAM was allocated to a second register file. Separate system and user register files do not directly affect BRISC CPU performance, but they do add complexity and parts to the BRISC CPU design and as a result may impact overall CPU performance. Complexity is added because ten registers of each register file are devoted to the global registers, which are shared by all register windows. In order to avoid addressing the global registers in the register file RAMs, additional RAMs are used (the *ADDSUB* module) to increment the Current Window Pointer (CWP). If the system registers were deleted, the global registers could be moved to the unused half of the register file RAM, counters could be used instead of the latches currently used in the CWP, and *ADDSUB* deleted. The system CWP and SWP could also be deleted, for a total chip savings of over 9 chips. CPU performance may be improved because of the smaller number of parts, but system performance may be degraded because of slower switching between system and user modes (but only if system mode required a separate stack from user mode).

REGISTER FILE: Since the register file is in the critical path, an interesting experiment would be to delete the register file entirely and replace it with a single set of registers as used in conventional architectures. Then register address calculation would be greatly simplified because the CWP would no longer be required and the register address determined only by the register number bits in the instruction. Deleting the register file would save at least two gate delays and associated wire delays for a total savings of at least 5 ns per phase (10 ns per cycle). A savings of 10 ns translates to a 21% performance increase for BRISC B. More than 30 chips would also be deleted from the design. The resulting CPU would be significantly faster and smaller, but calls, returns, and accesses to local variables would take many more instructions than required by the RISC I instruction set with the register file. More analysis needs to be performed to find the true cost of the register file.

REVERSE SUBTRACT: The *REVERSE SUBTRACT* instruction allows the two source operands of an instruction to be subtracted in the reverse order from the *SUBTRACT* instruction and is of questionable utility. *REVERSE SUBTRACT* has not been deleted because it does not seem to have any impact on the BRISC CPU performance or cost.

Design Additions

MULTIPLY: As will be shown in the benchmark section, BRISC performs well on benchmarks with multiplies, even without a *MULTIPLY* instruction. But BRISC could be made even faster if *MULTIPLY* could be added with little impact to the rest of the CPU. This could be accomplished by adding a single chip multiplier in parallel with the ALU. Currently available single chip multipliers are significantly slower than the BRISC CPU cycle time, but are getting faster. As faster multipliers become available, a multiplier will be a useful addition to the BRISC CPU.

FLOATING POINT: BRISC is not well suited for problems with heavy floating point emphasis because of the lack of floating point hardware in the BRISC CPU. Floating point performance of BRISC could be enhanced by adding hardware to do the time consuming tasks of floating point operations. This is in keeping with the RISC philosophy of adding hardware only for time consuming tasks. A possible addition for floating point is the multiplier mentioned in the previous paragraph. Another addition would be a leading zeroes counter to be used with the shifter for normalization. Ideally, changes for floating point should not complicate control, instead they should only enhance the data path for floating point operations. More research is required to identify the best way to add floating point to a RISC processor.

Redesign

MULTIPLEXORS VS. OPEN-EMITTER BUSSES: BRISC uses two methods for selecting one of several sources into an input. One is a multiplexor on the input; the other is an OR-gate for each of the sources with outputs tied together on an open-emitter bus. Examples of multiplexor input can be found in the *ALU PATH* and *SHIFTER PATH* inputs. An example of an open-emitter bus is the result bus *(RBUS)* which is driven by the Last PC, Current PC and the *RESULT LATCH*. Open-emitter busses have less logic delay, but can have longer wire delays because of the length of the busses and transmission line effects along the busses. Open-emitter busses should only be used if the sources are close together, or if there are too many destinations to add multiplexors on every input. More analysis may show that BRISC should use a different mix of multiplexors and open-emitter busses to increase performance or decrease cost.

SHARED MULTIPLEXORS: Bussing may also be improved by combining multiplexors. For example, analysis may show that the *SPECIAL REGISTER* input multiplexor may be combined with the *ALU PATH* input multiplexor for a savings of six parts. Sharing multiplexors may have an adverse affect on CPU performance because of longer wires caused by the sharing of a single multiplexor between two logical portions of the CPU.

REGISTER ADDRESS CALCULATION: As stated previously, register address calculation is a major contributer to the critical path of the BRISC CPU. Register address calculation should be redesigned to save gate and wire delays in the critical path. One or two gate delays may be

saved by reallocating some of the random logic used for register address calculation. The expected performance increase is between 2 and 5 ns per phase (4 to 10 ns per cycle), for a performance gain of between 9 and 21% for BRISC B.

3.1.2. Memory Speed

Section 3.1.1 derived the CPU cycle time for BRISC and suggested methods of improving the cycle time. CPU cycle time was derived for BRISCs A, B, and C by using three sets of assumptions for wire delay. The purpose of this section is to derive numbers for the cycle times of BRISCs A, B, and C weighted by the delays caused by the memory subsystem. These numbers are used to calculate the speed of BRISCs A, B, and C relative to a 400 ns RISC II processor. Performance of a 400 ns RISC II processor is not degraded by the memory subsystem because address translation and main memory access occur within a single 400 ns cycle, so no cache or TLB is required.

Memory subsystem performance is determined by many factors including cache performance, virtual memory performance, and access time of main memory. These factors interact with each other and with the CPU performance.

This section assumes that the overhead incurred by the virtual memory for BRISC is minimal. Virtual memory overhead includes virtual address translation time and page fault time. Virtual address translation time can be kept low with the use of a Translation Buffer (TLB). For example, the observed hit ratio for the Amdahl 470V/6 TLB is about 99.6 to 99.7%. [21] The penalty incurred by page faults can be kept low by using a large physical memory to keep the page fault rate low, and by switching processes during disk accesses caused by page faults to minimize their affect.

If virtual memory overhead is assumed to be low, then memory subsystem performance is determined by the cache performance and main memory access time. Factors affecting cache performance include cache size, line size, set size, cache bandwidth, main memory bandwidth, cache fetch algorithm, placement algorithm, replacement algorithm, write-through vs. write-back, cache priorities, and prefetch. Smith provides an excellent discussion of these factors. [21] Appendix B shows the variability of these factors for some typical commercial computers as presented by Smith [21], Pier [16] and Clark. [3]

For purposes of determining the memory subsystem performance, the cache performance factors mentioned in the previous paragraph may be summarized by three performance numbers: the access time of the cache, the cache hit ratio, and the memory waiting time due to cache misses. Access time of the cache is the time measured from the start of a memory request by the CPU to the time the request is fulfilled by the cache when there is a cache hit. Cache hit ratio is the percentage of CPU memory references that are found in the cache. Memory waiting time is

the average time that the CPU must wait for a cache miss. [3]

The access time of the cache should be designed to be at least as fast as the cycle time of the CPU, otherwise the CPU will never run at full speed. The 10K ECL RISC/E cache is the same speed as the 10K ECL RISC/E CPU; therefore, it is assumed that a 100K ECL cache may be built for BRISC with the same cycle time as the 100K BRISC CPU.

Best case performance for the memory subsystem is a hit rate of 100% so that the CPU always runs at full speed; BRISC A assumes a 100% hit rate as the upper bound on performance.

Figure 5 plots the affect of cache hit ratio and memory waiting time on BRISC B and BRISC C performance (Appendix C includes the data used to generate Figure 5). Three curves are shown for each processor, one each for 400, 800, and 1200 ns memory wait times. BRISC A with no memory wait time is also included for reference.

Smith [21] presents analytic results of cache hit ratios of over 99% for typical caches. He also presents empirical results of cache hit ratios of over 98% for user state programs on the Amdahl 470 with a 16K cache. Simulations performed by DEC for the PDP-11/70 cache showed a hit ratio of over 98% for a 2K cache with a line size of four words. [2] Based on these results BRISC B will assume a cache hit ratio of 98% as a reasonable estimate of cache performance.

Both the analytic and empirical curves presented by Smith seldom show less than a 95% hit ratio. DEC's simulations also show better than a 95% hit ratio for a 2K cache with a line size of one. As a worst case estimate, BRISC C will use a 95% hit ratio.

Memory waiting time may be approximated by the cycle time of main memory, as long as the line size equals the size of the path to main memory. Memory waiting time may be made less than the cycle time of main memory by using write buffers, but some of this gain may be lost because of interference from pre-fetches. Smith reports typical main memory access times of 300 to 600 ns for the Amdahl 470V/7 and IBM 3033. These numbers are similar to the main memory access times of smaller computers such as the Sun Workstation†, which has a main memory access time of 400 ns including virtual address translation time. [22] Clark reports a main memory access time of 1200 to 1400 ns for the VAX-11/780. [3] BRISC A is not affected by main memory access time since it has a 100% hit ratio; BRISC B assumes a 400 ns main memory access time and BRISC C assumes a 1200 ns main memory access time.

† *Sun Workstation* is a registered trademark of Sun Microsystems, Inc.

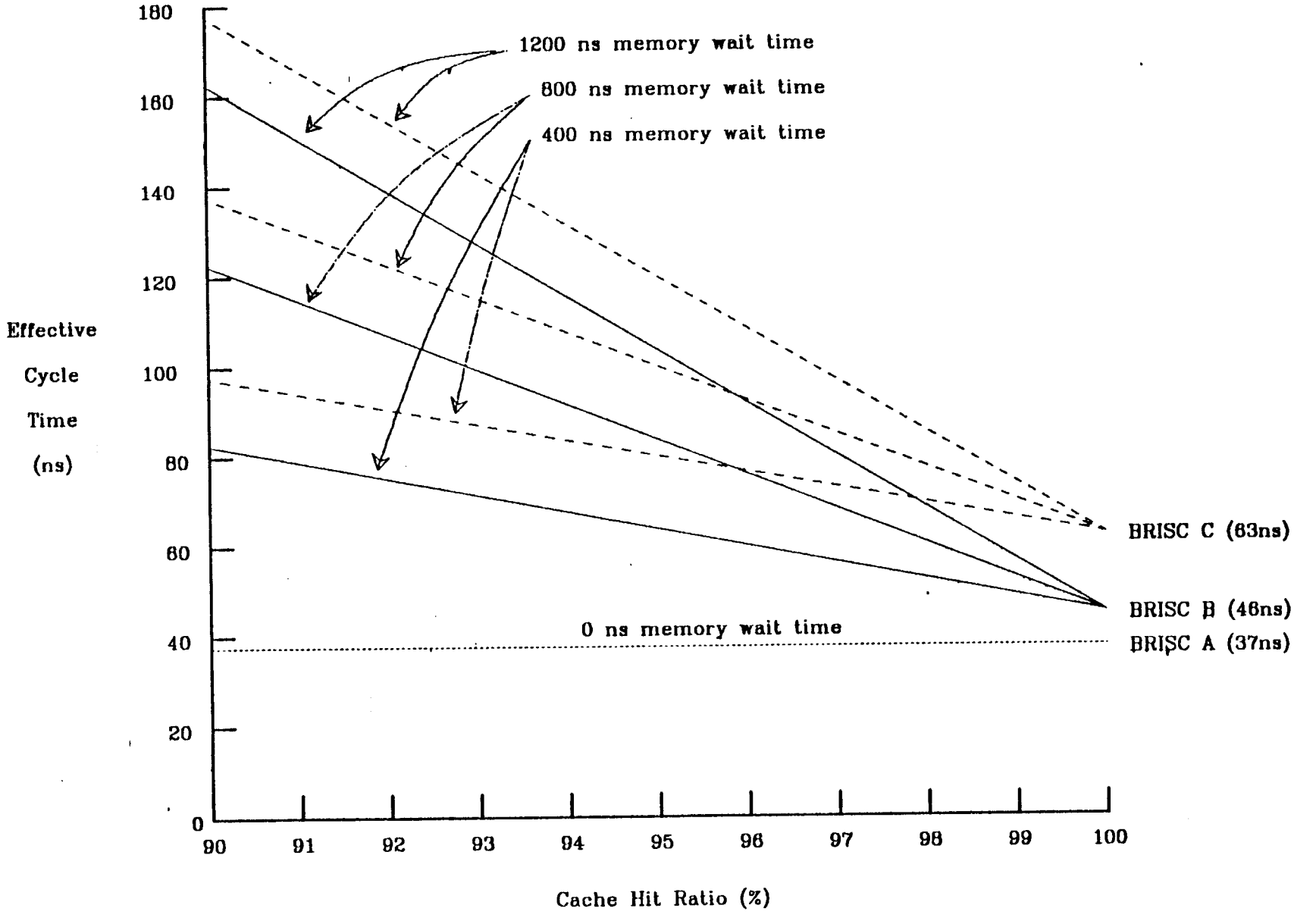


Figure 5. BRISC Effective Cycle Time

3.2. Benchmark Comparisons

Table 1 summarizes the performance of BRISCs A, B, and C based on the results of Section 3.1. The cycle times of 37, 53 and 120 ns were divided into the 400 ns cycle time of RISC II to determine the ratio of BRISC cycle time to RISC II cycle time, which is shown in the bottom line of Table 1. This ratio is used to calculate BRISC performance in the following benchmarks.

Table 1. BRISC A, B and C Performance Summary

FACTOR	BRISC A	BRISC B	BRISC C
Logic Cycle Time (ns)	37	37	37
Clock Skew (± ns)	0.0	0.1	0.1
Minimum Wire Delay (ns)	0.0	0.5	0.5
Maximum Wire Delay (ns)	0.0	1.0	2.0
TOTAL CPU CYCLE TIME (ns)	37	46	63
Cache Hit Ratio	100%	98%	95%
Miss Penalty (ns)	0	400	1200
EFFECTIVE CPU CYCLE TIME (ns)	37	53	120
BRISC/RISC II Cycle Time Ratio	.09	.13	.30

The original RISC II benchmarks were run with a RISC simulator. The RISC simulator takes into account the extra cycle required for loads and stores, and also takes into account the affect of register window spills into main memory. Three benchmarks are presented, the Puzzle program, TAK LISP benchmark, and UNIX Portable C Compiler. Three benchmarks are used to measure BRISC performance for different applications. Each benchmark presents the time in seconds to execute the benchmark. Benchmark performance is also presented as a multiple of VAX-11/780 performance, shown as the ratio of the number of seconds required by the VAX divided by the number of seconds required by the computer being measured. VAX performance is used because the VAX has become a pseudo-standard for comparison, and because a BRISC computer could be built for about the cost of a VAX.

Puzzle Program

The Puzzle program is a recursive puzzle solving program originated by Forest Baskett that has been used to benchmark many mainframe and minicomputers. Selinger [19] presents listings of the Puzzle program and sources of Puzzle benchmark data for many computers. Patterson and Séquin[14] identify benchmark results for a 400 ns RISC I. Table 2 lists the results of Puzzle benchmarks for BRISCs A, B and C relative to the Selinger and Patterson numbers. Table 2 only lists the better numbers where conflicting data is available for the same machine. BRISCs A and B outperform all available Puzzle times; even the worst case BRISC C outperformed all benchmarks except for the best time reported for the Amdahl 470V/8.

Table 2. Puzzle Benchmark Results					
MACHINE	TIME		LANGUAGE	COMMENTS	CPU COST
	sec	VAX sec/sec			
BRISC A (37 ns)	0.3	11.8	C	pointers	$132K
BRISC B (53 ns)	0.4	8.3	C	pointers	$132K
Amdahl 470V/8	0.7	5.0	C	pointers, reg vars	$1100K
BRISC C (120 ns)	1.0	3.6	C	pointers	$132K
Amdahl 470V/8	1.1	3.2	C	pointers	$1100K
Amdahl 470V/8	1.1	3.2	Pascal	no range checking	$1100K
Amdahl 470V/8	1.2	2.9	C	reg vars	$1100K
IBM 3081	1.4	2.5	Pascal		$3260K
Amdahl 470V/8	1.6	2.2	C	subscripts	$1100K
S-1 Mark 1	2.0	1.8	Pascal	subscripts	$2000K
Dorado	2.0	1.8	Mesa	pointers	$118K
Amdahl 470V/8	2.3	1.5	Pascal		$1100K
Dorado	3.0	1.2	Mesa	subscripts	$118K
RISC II (400 ns)	3.2	1.1	C	pointers	$10K
VAX 11/780	3.5	1.0	C	pointers	$132K
DEC 2060	4.4	0.8	Pascal	pointers	$502K
DEC 2060	4.6	0.8	C	pointers, hand opt	$502K
VAX 11/780	4.6	0.8	C	subscripts	$132K
RISC II (400 ns)	4.7	0.7	C	subscripts	$10K
DEC 2060	4.7	0.7	C	subscripts, hand opt	$502K
DEC 2060	5.3	0.7	C	pointers	$502K
DEC 2060	5.4	0.6	Pascal	subscripts	$502K
DEC KL10	6.0	0.6	C	pointers	$540K
VAX 11/780	6.1	0.6	Pascal	subscripts	$132K
PDP 11/70	6.4	0.5	C	pointers	$70K
DEC 2060	7.3	0.5	C	subscripts	$502K
IBM 158	7.5	0.5	Pascal	subscripts	$1550K
68000 (8 MHz)	17.0	0.2	C	pointers, nunix	$10K

TAK Benchmark

The TAK benchmark is a heavily recursive function used to measure the efficiency of LISP procedure calls as well as the efficiency of LISP fixnum and bignum arithmetic. [17] Fixnum arithmetic refers to integers of bounded length, whereas bignum arithmetic refers to integers of unbounded length. Figure 6 is a listing of the TAK benchmark. Table 3 presents the results reported by Ponder compared to the times calculated for BRISC. BRISCs A and B again outperform all available benchmarks; BRISC C is only beat by the Dorado.

Ponder estimated RISC I performance for LISP by compiling the LISP code on a VAX and hand translating the output to RISC I assembly code. Ponder states that a LISP compiler optimized for RISC could be built with better performance than is shown in Table 3. Recent studies by Ponder show that with minor enhancements a 400 ns RISC processor could be built that executed the TAK benchmark in under 0.66 seconds. This translates to times of 0.06, 0.09 and 0.20 seconds for BRISCs A, B, and C (5.6 to 18.0 times faster than a VAX).

```
(tak 18 12 6)

(defun tak(x y z)
      (cond ((not (lessp y x)) z)
            (t (tak (tak(sub1 x) y z)
                  (tak (sub1 y) z x)
                  (tak (sub1 z) x y)))))
```

Figure 6. TAK Benchmark

Table 3. TAK Benchmark Results				
MACHINE	TIME		LANGUAGE	CPU COST
	sec	VAX sec/sec		
BRISC A (37 ns)	0.2	5.9	PSL/Franz LISP	$132K
BRISC B (53 ns)	0.3	4.2	PSL/Franz LISP	$132K
Dorado	0.5	2.2	InterLISP	$118K
BRISC C (120 ns)	0.6	1.8	PSL/Franz LISP	$132K
DEC KL10	0.8	1.4	MacLISP	$540K
VAX 11/780	1.1	1.0	Franz LISP	$132K
RISC II (400 ns)	2.0	0.6	PSL/Franz LISP	$10K
68000 (8 MHz)	2.9	0.9	PSL SYSLISP	$10K
Dolphin	5.7	0.2	InterLISP	$40K

C Compiler

Miros ported the Portable C Compiler by Steve Johnson [8] to RISC I. [12] After porting the compiler, Miros compared the performance of the VAX Portable C Compiler running on a VAX to the VAX Portable C Compiler running on a 400 ns RISC I. His results are presented in Table 4. BRISCs A, B, and C are all significantly faster than a VAX.

Table 4. VAX Portable C Compiler Benchmark Results							
MACHINE	LD.C COMPILE TIME		SORT.C COMPILE TIME		PUZZLE.C COMPILE TIME		CPU COST
	sec	VAX sec/sec	sec	VAX sec/sec	sec	VAX sec/sec	
BRISC A (37 ns)	1.6	17.8	1.0	17.7	0.3	19.4	$132K
BRISC B (53 ns)	2.2	12.5	1.4	12.4	0.4	13.5	$132K
BRISC C (120 ns)	5.1	5.5	3.2	5.5	0.9	6.0	$132K
RISC II (400 ns)	16.9	1.7	10.6	1.6	2.9	1.8	$10K
VAX 11/780	27.9	1.0	17.4	1.0	5.2	1.0	$132K

Benchmark Summary

Figure 7 is a summary of some of the results from the above benchmarks. The best times are shown for four mainframes, two minicomputers, and two microcomputers. Performance is shown as a multiple of VAX performance. BRISCs A and B outperform all the listed computers, and BRISC C is competitive. Using BRISC B as the best estimate of BRISC performance, BRISC is 8 to 13 times faster than a VAX for C programs, and 4 times faster than a VAX for LISP programs.

4. Cost

Much as performance is determined by the combination of many factors, so is cost determined by many factors. Selinger defines computer workstation cost as the sum of the manufacturing and labor costs of parts, boards, cables, logic cages, backplanes, cabinets, power distribuction back panel, front panel, assembly, and test. [19] He then defines price as the sum of manufacturing cost, development cost, sales cost, service cost, and profit or loss. It is beyond the scope of this paper to estimate the values of all these items to calculate the price of a computer built around the BRISC CPU. Instead, the cost of a BRISC computer will be estimated by comparing BRISC to an existing computer to calculate the relative price of BRISC. The Dorado [9] computer by Xerox was selected as the existing computer for comparison because it is also an ECL computer, and is about the same size as a system built around BRISC.

The Dorado is a high-performance personal computer consisting of 3200 medium scale integrated components (not including memory), most of which are ECL 10K. [16] The Dorado contains everything a BRISC computer would need including up to 8 Megabytes of main memory, a high-performance cache with a 30 ns cycle time, peripheral interfaces, and a large 2050 watt power supply producing sufficient power and ECL voltages (-5V at 250A and -2V at 75A).

The cost for a BRISC computer is calculated by comparing the cost of the BRISC CPU to the Dorado CPU and keeping all other costs the same. The difference in cost of the two CPUs is found by first determining the difference in the number of chips. To compare the number of chips in the Dorado with the number of chips in BRISC, a lower bound is determined for the number of chips in the Dorado that would be replaced by the BRISC CPU.

The Dorado consists of up to 24 boards with up to 288 chips per board. Five of the 24 boards constitute the Dorado CPU and are listed in Table 5. Table 5 also lists a lower bound for the number of chips on each of the five boards in the CPU. [17]

About half of the Instruction Fetch Unit (IFU) is used to control the memory, the other half is devoted to the CPU; the IFU will not be considered as part of the CPU to obtain a conservative estimate.

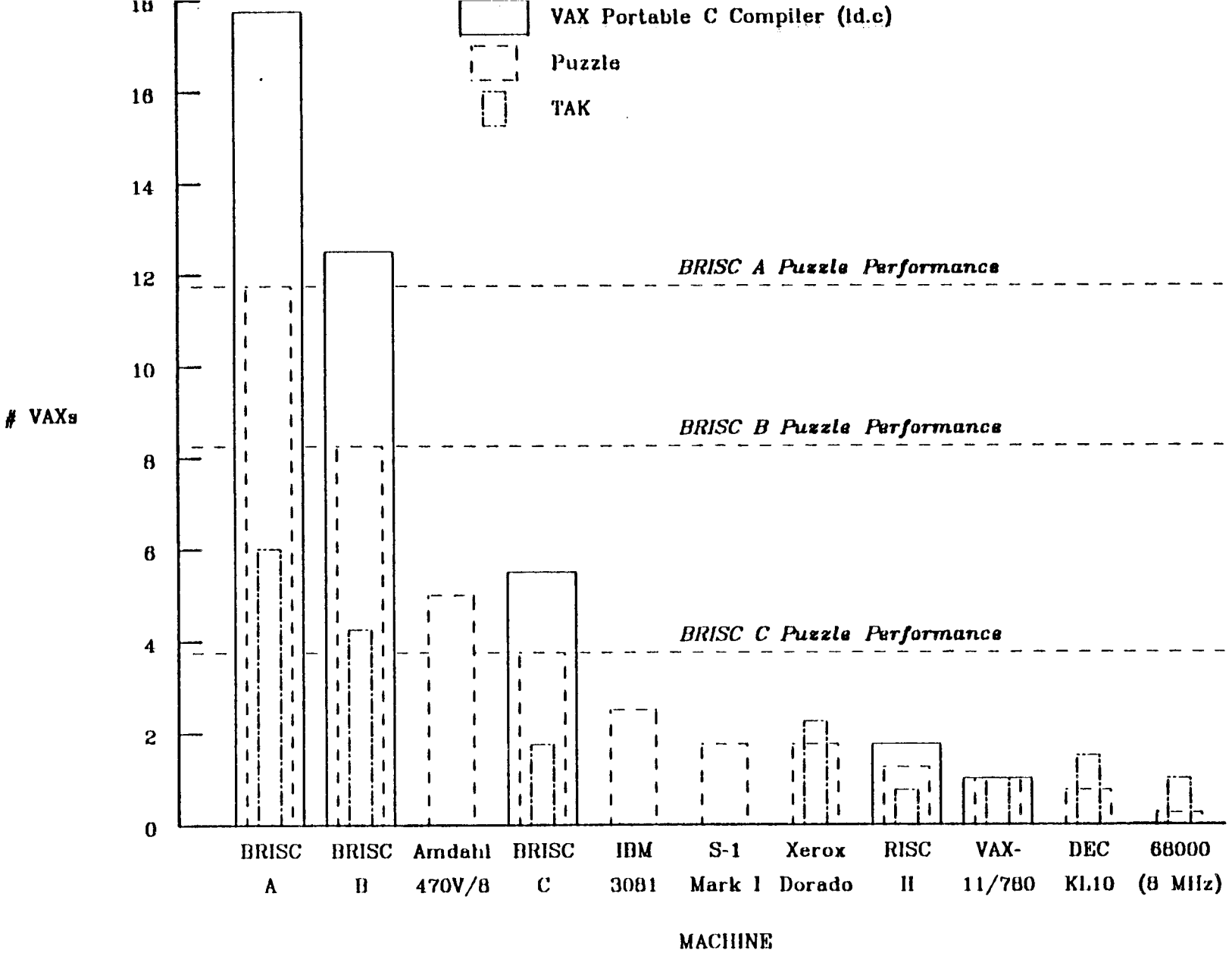


Figure 7. Performance Summary

Table 5. Dorado CPU Boards		
BOARD	% POPULATED	# CHIPS (288 Max)
Instruction fetch unit	80%	230
Processor high byte	90-95%	260-275
Processor low byte	90-95%	260-275
Control section	90-95%	260-275
Microinstruction memory	90-95%	260-275

The Dorado CPU performs task switching at the microcode level. Microcode tasks are called *microtasks*. Some microtasks are used as device controllers - a DMA controller would therefore be required if the Dorado CPU was removed. A DMA controller would be no more complex than the Control Section of the Dorado, so the Control Section will not be counted as part of the Dorado CPU.

Ignoring the IFU and Control Section leaves three boards in the Dorado CPU: the Processor High Byte, Processor Low Byte, and the Microinstruction Memory. These three boards contain about 780 chips. The 332 100K chips of the BRISC CPU would occupy the same board space required by about 664 10K chips because of the 24-pin packages used by the 100K parts versus the 16-pin packages used by the 10K parts. The BRISC CPU therefore requires 15% less board area than the Dorado CPU. Insertion cost is also 15% less for BRISC; 24-pin packages cost twice as much to insert as 16-pin packages, [19] but BRISC has 43% the number of chips of the Dorado CPU.

In order to calculate a conservative estimate for the cost of a BRISC CPU, the Dorado and BRISC CPUs are assumed to have the same PC board and IC insertion costs. PC board and insertion costs being equal, the difference in cost of the two CPUs will therefore be largely determined by the difference in parts cost. Parts cost of the Dorado CPU is not known (except by Xerox). The least expensive 10K part sells for 61 cents in quantities of 100 to 1000, so a lower bound on parts cost for the Dorado CPU is $475.80. The BRISC parts list (Appendix D) shows the parts cost of BRISC is $3330.32 in parts quantities of 100 to 1000. The resulting parts cost difference between the BRISC CPU and the Dorado CPU is at most $2854.52. Selinger calculated the markup for the VAX-11/780 to be 5.2 times, which he showed to be consistent with the markups used by other manufacturers. Using a 5.2 times markup for the parts differential between BRISC and Dorado gives a selling price difference of $14,843.50. Dorados sell for $129,500 with 2 Megabytes (MB) of main memory and an 80 MB disk. Discounting for the cost of an 80 MB disk ($11,900) [23] gives a selling price of $117,600 for the Dorado without peripherals. A BRISC computer could therefore be sold for less than $14,843.50 over the $117,600 selling price of a Dorado, or about $132,444. This is nearly identical to the price of a VAX-11/780 without peripherals, which Selinger calculates to be $132,400 with 1 MB of main memory (the Dorado comes with 2

MB).

5. Closing Remarks

This section presents retrospectives by the BRISC designers on the BRISC project.

5.1. Comments on SCALD

The BRISC project would not be at the point it is today without the SCALD system. SCALD has proven to be an invaluable tool for computer design. Traditionally, computer design has been a rigid process because of the difficulty of making changes to hardware once it is built. SCALD provides a vehicle to do *exploratory hardware design*, much as a good programming environment allows *exploratory programming*. Bean Sheil defines exploratory programming as an approach to programming combining system design with implementation. [20] In an exploratory programming environment, the programmer experiments with changes to the design and immediately sees their effect on the operation of the system. Exploratory hardware design combines design and implementation in the sense that the hardware designer can make changes to the hardware design and immediately see the effect on the operation of the hardware system. Effects are seen by using timing verification, simulation and post processing rather than by building and testing physical hardware. In this way many possible solutions to the same problem may be tried and evaluated, and the best solution selected. The best solution is picked on the basis of logical correctness, speed and parts cost.

SCALD provides the equivalent of what Sheil calls *programming power tools* in the form of five application programs: a graphics editor, compiler, timing verifier, simulator, and post processor. These tools amplify the power of the hardware designer so that he can produce a better design in less time.

As with most software tools, the tools provided by the Valid SCALD system have several deficiencies that detract from the usefulness of the system, the most noteable being the speed of the system. This section describes the speed of the Valid SCALD system and comments on each of the Valid SCALD tools from the point of view of the BRISC designers.

5.1.1. SCALD Speed

While SCALD comes a long way in making exploratory hardware design as simple as exploratory programming, the single biggest detractor is the speed of the SCALD system. Exploratory hardware design implies interactive use of the SCALD system, which means that fast response is required for most commands (i.e. less than one second), and response time may occasionally go up to 30 seconds. Studies have shown that user productivity goes down when response times go over three seconds, probably because of the disruption of user thought processes. [24]

Table 6 lists a few of the response times encountered during the BRISC design. While Valid has significantly improved the speed of the post processor - it once took over six hours for BRISC with 606 parts, and now only takes 12 minutes with 332 parts - the response times need to be much better to allow true interactive design.

Table 6. Valid SCALD Speed

FUNCTION	APPLICATION PROGRAM	MODULE	ELAPSED TIME (min:sec)	CPU TIME (min:sec)
Timing Verification	Compiler	PCs & Address	3:58	3:23
		BRISC	23:19	22:21
	Timing Verifier	PCs & Address	2:31	2:30
		BRISC††	13:32	12:19
Simulation	Compiler	PCs & Address	4:13	3:37
		BRISC	32:26	26:56
	Simulator (start-up)	PCs & Address	7:10	
		BRISC	17:59	
	Simulator (per cycle)	PCs & Address	:04	
		BRISC	:36	
Post Processing	Compiler	PCs & Address	1:59	1:28
		BRISC	10:19	9:34
	Post Processor	PCs & Address	1:40	1:38
		BRISC	12:23	12:19

††Does not include PCs & Address, Cache, Special Registers, or Result Latch.

In fairness to Valid it should be mentioned that the Valid SCALD system used for BRISC runs on a 68000. Valid also sells SCALD systems for VAXs (with the VMS operating system) and IBM 370s. Table 2 may be used to compare the performance and cost of the 68000, VAX and IBM 370 (Models 158 and 3081).

5.1.2. Graphics Editor

The graphics editor proved to be a useful tool for entering drawings and changing drawings after they were entered. The graphics editor is very fast for most drawings and encourages interactive design. The graphics editor's interactive nature made it much more useful than other drawing tools previously available at Berkeley.

5.1.3. Compiler

The compiler converts drawings from the format created by the graphics editor to a format usable by the timing verifier, simulator and post processor. As such, the compiler is nothing more than an intermediary between the graphics editor and the other applications as opposed to a design tool.

The compiler has proven useful for finding assertion violations. Assertion violations are signals that are asserted high which are inadvertently tied to signals that are asserted low. While only a few such errors have been found by the compiler in BRISC, the errors were detected faster and corrected sooner than if they had been found with the simulator (see Table 6).

The compiler is not designed for interactive use. A command file must be edited every time a different type of output is required (i.e. for the timing verifier, simulator or post processor), and every time a different drawing is compiled. The command file supports batch versions of SCALD, and is not appropriate for an interactive environment.

5.1.4. Timing Verifier

The timing verifier uses minimum and maximum timing parameters for all parts in a circuit to calculate a value history of all signals during a single clock cycle. The timing verifier has two uses, one is to find timing violations such as changing signals violating set-up times, the other is to calculate the cycle time of a circuit.

Timing Violations

The timing verifier is useful when it finds valid timing violations, but is sometimes difficult to use and is too conservative. To get realistic outputs, timing behavior of undriven inputs to a circuit must sometimes be specified, otherwise all signals may stay stable and no verification will occur. Sometimes, the timing verifier is so conservative that it finds errors that could never occur in a real circuit. For example, the timing verifier once identified the input to a flip-flop to be changing before the clock, even though the input was driven by the negative output of the flip-flop (that cannot possibly change before the clock).

Some timing violations cause the timing verifier to go into an endless loop, making them exceedingly difficult to find. A few timing violations have been detected in BRISC by the timing verifier; mostly with the endless loop method.

Cycle Time Calculation

The timing verifier does not automatically identify the worst case path through a circuit, as is done by Crystal. [12] Instead, the designer must examine the value history of all signals in a circuit and find the signal that is stable last. The worst case path is important for two reasons. First, the delay through the worst case path must be known to find the minimum cycle time of a circuit. Second, the designer should concentrate on optimizing the worst case path to speed up the circuit, but he needs to know the worst case path before he can fix it. The minimum cycle time may also be found by running the timing verifier with successively shorter and shorter clock cycles until timing errors are found; that still does not identify the worst case path and is time

consuming.

Since large CPUs tend to spend about half of their cycle times in wire delays, it is essential that wire delays be included in timing verifier calculations. Wire delays for ECL should include both delays due to the length of the wires, and delays induced by loads along the wires. Because of the speed of ECL, most wires act like transmission lines, and the wire delay calculator should treat them as such. As was mentioned in Section 3.1.1.1, a single pair of minimum and maximum wire delays may be specified to the verifier, but that is too crude an approximation. Alternatively, the timing verifier does have the capability to accept calculated wire delays from a physical design system, but the Valid SCALD system does not include a physical design system. Even though a SCALD design may be transformed into the input format of another vendor's design system, this precludes exploratory hardware design because it is no longer interactive; the design cycle is measured in days instead of minutes when two design systems are used.

5.1.5. Simulator

The simulator does logic simulation of a circuit to verify logical correctness of a design. The simulator is potentially the most useful of the SCALD tools, but it also takes the longest to get any results and to correct errors that are found. The simulator has been used to simulate the operation of the *PCs AND ADDRESS*, the *ALU PATH*, and the *SHIFTER PATH*. These modules were picked for simulation because they each do a well defined logical function. The entire BRISC design has been loaded into the simulator, but simulation has only been partially completed because of lack of time.

Once the simulator is started, it is difficult to use because every signal of interest must be identified for display, and most inputs initialized. A batch file may be used to identify and initialize signals, but that seems to defeat the interactive capability of the simulator, and the batch file is just as difficult to generate in the first place. The simulator is generally used to trace the values of signals through progressive stages of logic. To do this more and more signals must be displayed and the circuit stepped a single cycle at a time to trace the behavior of signals. When an error is found, the simulator must be exited, the drawing edited to fix the error, the drawing recompiled, and the whole tedious simulation process restarted to get to the same point. A batch file may be used to return to the same point in the simulator, but if the error was fixed many of the signals being displayed are probably no longer needed because they were only displayed to find the error.

To provide exploratory hardware design, the simulator should interact with the graphics editor to allow graphical identification of signals of interest. Incremental recompilation should be provided to allow rapid update. Identification and correction of errors should be accomplished without leaving the simulator. These capabilities are identical to the debugging capabilities

provided by good exploratory programming environments such as Smalltalk-80‡. [6]

Even though it is hard to use, the simulator has found about a dozen errors in BRISC that would have prevented correct operation of the BRISC CPU.

5.1.3. Post Processor

The post processor has been used to find loading errors, undriven signals, and to count the number of ICs (physical packaging) in the BRISC CPU.

Loading Errors

The post processor finds loading errors by calculating the fan-out of every logical part and flagging those parts that exceed fan-out limits. This function of the post processor found about 20 loading errors in the BRISC design, all have since been corrected.

Undriven Signals

The post processor identifies all nets in a design and flags nets that are not driven. This is a valuable function because on a large design such as BRISC it is easy to assume that a signal is going to be generated and then forget to generate it or accidentally use a different name. Some nets are intentionally not driven, such as the reset input that is assumed to be externally generated. The post processor found over 15 signals that were inadvertently left undriven in the original BRISC design, which have since been corrected.

Physical Packaging

The physical packager assigns logical parts to physical parts and gives a parts count of each type of part. This is useful for finding the cost of the design, and to verify that the design will fit in the allotted space (e.g. a single circuit board). Exploratory design with the graphics editor and physical packager reduced the BRISC parts count from 606 to 342 for a savings of 44% as described in Section 1.2.

5.2. Lessons Learned

The BRISC project has been instructive to the BRISC designers in two areas: computer architecture and Computer Aided Design.

‡ *Smalltalk-80* is a registered trademark of Xerox Corp.

Computer Architecture

BRISC has shown us that the RISC concepts for computer design can be successfully applied to the design of high performance computers for integer high level language programs. We have found that it pays to concentrate effort on the data path to shorten the cycle time. We have also found that the smaller control implied by RISC helps to greatly reduce the parts count, decrease the cycle time and decrease the design time. The smaller parts count helps to keep the cycle time down by minimizing both logic delays and wire delays. Furthermore, the simple control used by BRISC prevents pipeline flushes so that cycles are not wasted.

Computer Aided Design

We have learned that designing computers without computer aided design is like writing programs without compilers. Computer aided design with SCALD allows design iterations for correcting design errors and for tuning the design without the expense and inconvenience of building hardware. The SCALD system by Valid has the functions needed for such exploratory design, it just needs more speed, fewer bugs and a physical design system. Even with these few weaknesses, there is no way we could have done this project without the Valid SCALD system.

The slowness of the SCALD system may be partially due to the speed of the 68000 CPU used by Valid. SCALD is a good example of an integer high level language application that could be enhanced by a BRISC processor.

5.3. Future Work

The two designers of BRISC are both interested in continuing the BRISC project. Work needs to be done in all aspects of the project described in this paper: hardware design, timing verification, simulation, and post processing. Once these are completed, then BRISC will be ready for fabrication.

Hardware Design

The hardware design for BRISC is complete except for a couple of minor portions of the CPU such as a comparator to detect register-as-memory accesses to the register file. The interface to the support processor needs to be designed, and the 'microcode' must be completed for all instructions.

Even though the design is nearly complete, we would like to make some changes to the design to speed it up and reduce parts count. One area that could benefit from further redesign is control. While the current implementation of control is logically correct, extra bits have been included for compatibility with old versions of the microcode and to aid in debugging. Before

fabrication, control should be reorganized to remove unneeded bits. The resulting design should be smaller than the current design (by at least five chips).

Timing Verification

We feel confident that the worst case path has been timing verified and that we have a worst case estimate for the speed of BRISC. More timing verification is needed to find timing errors. Furthermore, accurate wire delays should be used to get a more accurate estimate of the BRISC cycle time.

Simulation

Simulation has been completed on key portions of BRISC, but needs to be performed on the entire design. Our goal is to use SCALD to deposit programs into the simulated memory and simulate BRISC executing those programs. Such a simulation would convince us that BRISC is ready for fabrication.

Post Processing

Further post processing will be done to reduce the parts count and to prepare the BRISC design for fabrication.

Fabrication

One design goal for BRISC has been to limit the parts count to fit the BRISC CPU onto a single S1 Mark IIA wire-wrap board. The S1 wire-wrap board is about 24 by 24 inches and has space for 325 100K ECL ICs with associated termination resistors and bypass capacitors. [5] The current 332 chip design of BRISC can be modified to fit on the S1 board with some minor redesign.

BRISC could be made faster by using a denser board that leads to shorter wire delays. A new board being developed at Livermore for the S1 Mark III is such a board. The Mark III board is 16 by 16 inches and has space for 687 100K ECL ICs (on custom carriers). [3] The extra space could be used for the cache. A disadvantage of the Mark III board is that it must be water cooled, so it would only be appropriate if BRISC was imbedded in a larger system.

Another fabrication technique would be to redesign BRISC using 100K gate arrays. The resulting design would require fewer chips and would be faster because of shorter wire delays, but would be more difficult to debug and change than discrete logic. A discrete logic version of of the design could be built as a prototype prior to the gate array version to debug the design.

6. Conclusion

BRISC has shown that a RISC architecture developed for a single chip processor can be successfully applied to a discrete logic CPU. The resulting processor can be developed for the cost of a large minicomputer (about $132K) and yet outperforms mainframes costing millions of dollars on integer high level language programs. In addition, the resulting processor can be designed quickly because of the simplified architecture and because of the power of computer aided design tools such as SCALD. As a result, the RISC I architecture can serve as the basis for a family of processors, starting with medium performance single chip processors such as RISC II and progressing up to high performance discrete logic processors such as BRISC. Our experience leads us to believe that the RISC style of architecture will flourish with advances in density and speed of new implementation technologies.

7. Acknowledgements

First, I must thank my partner Jeff Deutsch. RISC CPUs tend to make good two person projects, and Jeff contributed much needed talents to our team. He designed half of BRISC and wrote half of our report for the SCALD class during Winter 1983. Parts of our SCALD report have been incorporated into this paper, including some written by Jeff. Many of Jeff's ideas appear throughout this papers, including the multiplication and floating point enhancements to BRISC.

BRISC would not have been possible without my advisor, Dave Patterson. He helped to start the RISC project at Berkeley and provided guidance and direction for both the BRISC project and this report. I would also like to thank my second faculty reader, John Ousterhout, who agreed to read this report on short notice even though he had other commitments at the time.

I would like to thank all those involved with the RISC project for making BRISC possible. I would especially like to thank those involved with the RISC/E design study: Lloyd Dickman, Scott Baden, Jim Beck, Paul Hansen and Michael Shiloh. Special thanks go to the designers of RISC/E, Jim Beck and Helen Davis. Both Jim and Helen were very helpful in explaining the RISC/E design, and Helen was a great help throughout the project by patiently answering all of my questions on RISC and RISC/E. Many ideas throughout this paper are due to Helen's suggestions, including deletion of the register file, redesign of register address calculation, and redesign of the bussing structure by sharing multiplexors.

I would also like to thank Glen Miranker for explaining the SCALD system, and Valid Logic Systems for donating their equipment and time to the EECS department at Berkeley.

I am indebted to all those who proofread this paper (in alphabetical order): my parents (Art and Maria Blomseth), my uncle (Bob Blomseth), Helen Davis, Pete Foley, John Ousterhout, and

Dave Patterson. They all provided excellent comments, any problems left in the paper are my own fault.

Most of all, I would like to thank my wife, Ginny, and son, Kevin, for putting up with me going off to school and giving me emotional support along the way.

References

1. Stone, Harold S., Tien Chi Chen, Michael J. Flynn, Samuel H. Fuller, William G. Lane, Herschel H. Loomis, Jr., William M. McKeeman, Kay B. Magleby, Richard E. Matick, and Thomas M. Whitney, *Introduction to Computer Architecture*, Science Research Associates, Inc..
2. Bell, C. Gordon, J. Craig Mudge, and John E. McNamara, *Computer Engineering - A DEC View of Hardware Systems Design*, Digital Press, Bedford, Massachusetts, 1978.
3. Clark, Douglas W., *Cache Performance in the VAX-11/780*, Systems Architecture Group, Digital Equipment Corporation, Tewksbury, MA 01876, March 1982.
4. Davidson, Howard L., *Personal Communication*, July, 1983. Lawrence Livermore National Laboratory
5. Dickman, Lloyd, Scott Baden, Jim Beck, Paul Hansen, and Michael Shiloh, *RISC/E: Extended Performance Processor Design Study*, Computer Science Division, Department of E.E.C.S, University of California, Berkeley, CA, 1982.
6. Farmwald, Michael, *Personal Communication*, July, 1983. Lawrence Livermore National Laboratory
7. Goldberg, Adele and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company, Menlo Park, CA, 1983.
8. Johnson, S.C., "A Portable Compiler: Theory and Practice," in *Proc. Fifth Annual ACM Symposium of Programming Languages*, pp. 97-104, Tucson, Arizona, January 1978.
9. Lampson, Butler W. and Kenneth A. Pier, "A Processor for a High-Performance Personal Computer," in *Proceedings of the 7th Annual Symposium on Computer Architecture*, IEEE, May 6-8, 1980.
10. Mayo, Robert N., John K. Ousterhout, and Walter S. Scott, *1980 VLSI Tools, Selected Works by the Original Artists*, Computer Science Division, Department of E.E.C.S, University of California, Berkeley, CA, April 12, 1980.
11. McWilliams, Thomas M. and Lawrence C. Widdoes, Jr., *SCALD: Structured Computer-Aided Logic Design*, Computer Science Department, Stanford University and Lawrence Livermore Laboratory, University of California, 1982.
12. Miros, James C., *A C Compiler for RISC I*, Computer Science Division, Department of E.E.C.S, University of California, Berkeley, CA, 1982.
13. Ousterhout, John, *Using Crystal for Timing Analysis*, Computer Science Division, Department of E.E.C.S, University of California, Berkeley, CA, 1983.
14. Patterson, David A. and Carlo H. Séquin, "A VLSI RISC," *Computer*, vol. 15 , no. 9, pp. 8-21, September 1982.
15. Picha, Marianne, *DAPL: A General Purpose Microprogramming Assembler for UNIX*, Computer Science Division, Department of E.E.C.S, University of California, Berkeley, CA, October 2, 1980.
16. Pier, Kenneth A., "A Retrospective on the Dorado, A High-Performance Personal Computer," in *Proceedings of the 10th Annual Symposium on Computer Architecture*, IEEE, 1983.
17. Pier, Kenneth A., *Personal Communication*, July, 1983. Xerox Corporation
18. Ponder, Carl, *'...but will RISC run LISP??' (a feasibility study)*, Computer Science Division, Department of E.E.C.S, University of California, Berkeley, CA, April 8, 1983.
19. Selinger, Robert David, *The Design and Evaluation of Single and Multiple Processor Office Workstations*, PhD Dissertation, Computer Science Division, Department of E.E.C.S, University of California, Berkeley, CA, June 9, 1983.

20. Sheil, Beau, "Environments for Exploratory Programming," *Datamation*, February 1983.

21. Smith, Alan Jay, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, pp. 473 - 530, September 1982.

22. Sun Microsystems, Inc., *Sun Workstation Architecture*, Sun Microsystems, Inc., Mountain View, CA, Feb 1983.

23. Sun Microsystems, Inc., *U.S. and Canadian Price List*, Sun Microsystems, Inc., Mountain View, CA, April 15, 1983.

24. Thadhani, A. J., "Interactive User Productivity," *IBM Systems Journal*, vol. 20, no. 4, 1981.

25. UC Berkeley, *RISC I Principles of Operation*, Computer Science Division, Department of E.E.C.S, University of California, Berkeley, CA, Oct 1981. draft

APPENDIX A. BRISC INSTRUCTION SET

Appendix A
BRISC Instruction Set

Instr.	Operands	Comments	
ADD	S1,S2,D	D ← S1 + S2	integer add
ADDC	S1,S2,D	D ← S1 + S2 + carry	add with carry
SUB	S1,S2,D	D ← S1 - S2	integer subtract
SUBC	S1,S2,D	D ← S1 - S2 - carry	subtract with carry
RSUB	S1,S2,D	D ← S2 - S1	reverse integer subtract
RSUBC	S1,S2,D	D ← S2 - S1 - carry	subtract with carry
AND	S1,S2,D	D ← S1 & S2	logical AND
OR	S1,S2,D	D ← S1 | S2	logical OR
XOR	S1,S2,D	D ← S1 xor S2	logical EXCLUSIVE OR
SLL	S1,S2,D	D ← S1 shifted by S2	shift left logical
SRL	S1,S2,D	D ← S1 shifted by S2	shift right logical
SRA	S1,S2,D	D ← S1 shifted by S2	shift right arithmetic
LOADW	S1(S2),D	D ← M[S1+ S2]	load word
LOADWR	#L,D	D ← M[pc+ #L]	load word relative
LOADWS	S1(S2),D	D ← M[S1+ S2]	load word special
LOADHU	S1(S2),D	D ← M[S1+ S2]	load halfword unsigned
LOADHUR	#L,D	D ← M[pc+ #L]	load halfword unsigned relative
LOADHUS	S1(S2),D	D ← M[S1+ S2]	load halfword unsigned special
LOADHS	S1(S2),D	D ← M[S1+ S2]	load halfword signed
LOADHSR	#L,D	D ← M[pc+ #L]	load halfword signed relative
LOADHSS	S1(S2),D	D ← M[S1+ S2]	load halfword signed special
LOADBU	S1(S2),D	D ← M[S1+ S2]	load byte unsigned
LOADBUR	#L,D	D ← M[pc+ #L]	load byte unsigned relative
LOADBUS	S1(S2),D	D ← M[S1+ S2]	load byte unsigned special
LOADBS	S1(S2),D	D ← M[S1+ S2]	load byte signed
LOADBSR	#L,D	D ← M[pc+ #L]	load byte signed relative
LOADBSS	S1(S2),D	D ← M[S1+ S2]	load byte signed special
LOADIH	#L,D	D<31:13>←#L; D<12:0>←0	load immediate high
LOADIR	#L,D	D ← M[pc+ #L]	load immediate relative
STOREW	S,D1(D2)	M[D1+ D2] ← S	store word
STOREWR	S,#L		M[pc+ #L] ← S
STOREWS	S,D1(D2)	M[D1+ D2] ← S	store word special
STOREH	S,D1(D2)	M[D1+ D2] ← S	store halfword
STOREHR	S,#L		M[pc+ #L] ← S
STOREHS	S,D1(D2)	M[D1+ D2] ← S	store halfword special
STOREB	S,D1(D2)	M[D1+ D2] ← S	store byte
STOREBR	S,#L		M[pc+ #L] ← S
STOREBS	S,D1(D2)	M[D1+ D2] ← S	store byte special

Appendix A
BRISC Instruction Set (continued)

Instr.	Operands	Comments	
JMP	COND,S1(S2)	next pc ← S1+S2	conditional jump
JMPR	COND,#L	pc ← next pc + #L	conditional jump relative
CALL	D,S1(S2)	D ← pc; next pc ← S1+S2, CWP−	call
CALLR	D,#L	D ← pc; next pc ← pc+#L, CWP−	call relative
RET	S1(S2)	pc ← S1+S2, CWP++	return
TRAP	D	D ← pc; next pc ← trap vector, CWP−	trap
CALLINT	D,#V	D ← last pc; next pc ← #V; disable interrupts	hardware interrupt
RETINT	S	pc ← S; CWP++; enable interrupts	return from interrupt
GETCPC	D	D ← pc	get current pc
GETLPC	D	D ← last pc	get last pc
GETCWP	D	D ← CWP	get current window pointer
GETSWP	D	D ← SWP	get saved window pointer
GETPSW	D	D ← PSW	load status word
PUTCWP	S	CWP ← S	put new current window pointer
PUTSWP	S	SWP ← S	put new saved window pointer
PUTPSW	S	PSW ← S	put new status word

(1) *S*, *S1*, and *S2* are source registers and specify one of *R0* through *R31*. R0 is always zero. S2 may also be a 13-bit immediate value specified as #*CONSTANT*.

(2) *D*, *D1*, and *D2* are destination registers and specify one of *R0* through *R31*. D2 may also be a 13-bit immediate value specified as #*CONSTANT*.

(3) #*L* is a 19-bit immediate value.

(4) #*V* is a hardware generated interrupt vector.

(5) *COND* is a jump condition and specifies one of:

none	jump always (unconditional jump)
ne	jump if not equal
eq	jump if equal
nc	jump if no carry
c	jump if carry
nv	jump if no overflow
v	jump if overflow
lt	jump if less than
le	jump if not greater than
gt	jump if greater than
ge	jump if not less than
los	jump if lower or same
hi	jump if higher
p	jump if plus (positive)
m	jump if minus
no	jump never

APPENDIX B. COMMERCIAL CACHE SUMMARY

Appendix B
Commercial Cache Summary

MACHINE	SET SIZE (words)	# SETS	LINE SIZE (bytes)	CACHE SIZE (bytes)	NOTES
Amdahl 470V/6	2	256	32	16K	1, 3, 4
Amdahl 470V/7	8	128	32	32K	1, 3, 4, 14
Amdahl 470V/8	4	512	32	64K	1, 3, 4, 14, 16
Dorado	2	256	16	8K	5, 6, 15, 19
Honeywell 66/60	4	128	16	8K	2
Honeywell 66/80	4	128	16	8K	2
IBM 4331				4K	
IBM 4341				16K	
IBM 370 158-1	2				
IBM 370 158-3	4				
IBM 370 168-1	4	128	32	16K	2, 12
IBM 370 168-3	8	128	32	32K	2, 12
IBM 3033	16	64	64	64K	2, 7, 8, 12, 14, 19
IBM 3081D			128	32K	1, 8
IBM 3081K			128	64K	1, 8
Itel AS/6	4	128	32	16K	2, 8
Magnuson M80/42				16K	
Magnuson M80/43				32K	
Magnuson M80/44				16K	
NEC ACOS 9000				128K	
PDP-11/70	2	256	4	1K	2, 9
RISC/E	2	64-1024	16-128	8K	1, 11
S1 Mark IIa	4	5120	72	90K	1, 11, 17, 18
VAX-11/750	2			4K	2, 6, 10, 15
VAX-11/780	2	512	8	8K	2, 6, 10, 15

Note 1: Write back.

Note 2: Write through.

Note 3: Hit ratio over 98% measured in user mode, over 95% in supervisor mode.

Note 4: 4-byte data path between CPU and cache.

Note 5: Hit ratio over 99% measured.

Note 6: No write allocate.

Note 7: No reorder.

Note 8: 8-byte data path between CPU and cache.

Note 9: 2-byte data path between cache and memory.

Note 10: 4-byte data path between cache and memory.

Note 11: LRU allocation.

Note 12: Modified LRU allocation.

Note 13: 85%-90% hit ratios measured (with 30 to 40 users active), 95%-99% hit ratios calculated.

Note 14: Fetch bypass.

Note 15: Prefetch.

Note 16: Prefetech on miss.

Note 17: Virtual address cache.

Note 18: Multiprocessor support using cache coherence.

Note 19: One write buffer.

APPENDIX C. EFFECTIVE CYCLE TIME CALCULATION

Appendix C
Effective Cycle Time Calculation

PROCESSOR	MEMORY WAIT TIME (ns)	EFFECTIVE CYCLE TIME (ns)	
		100% cache hit	90% cache hit
BRISC A	400	37	73
	800		113
	1200		153
BRISC B	400	46	81
	800		121
	1200		161
BRISC C	400	63	97
	800		137
	1200		177

Only two values for hit ratios are shown because hit ratio versus effective cycle time is a linear function. The function used to calculated the effective cycle time is:

effective cycle time = (hit ratio)(cpu cycle time) + (1 - hit ratio)(memory wait time)

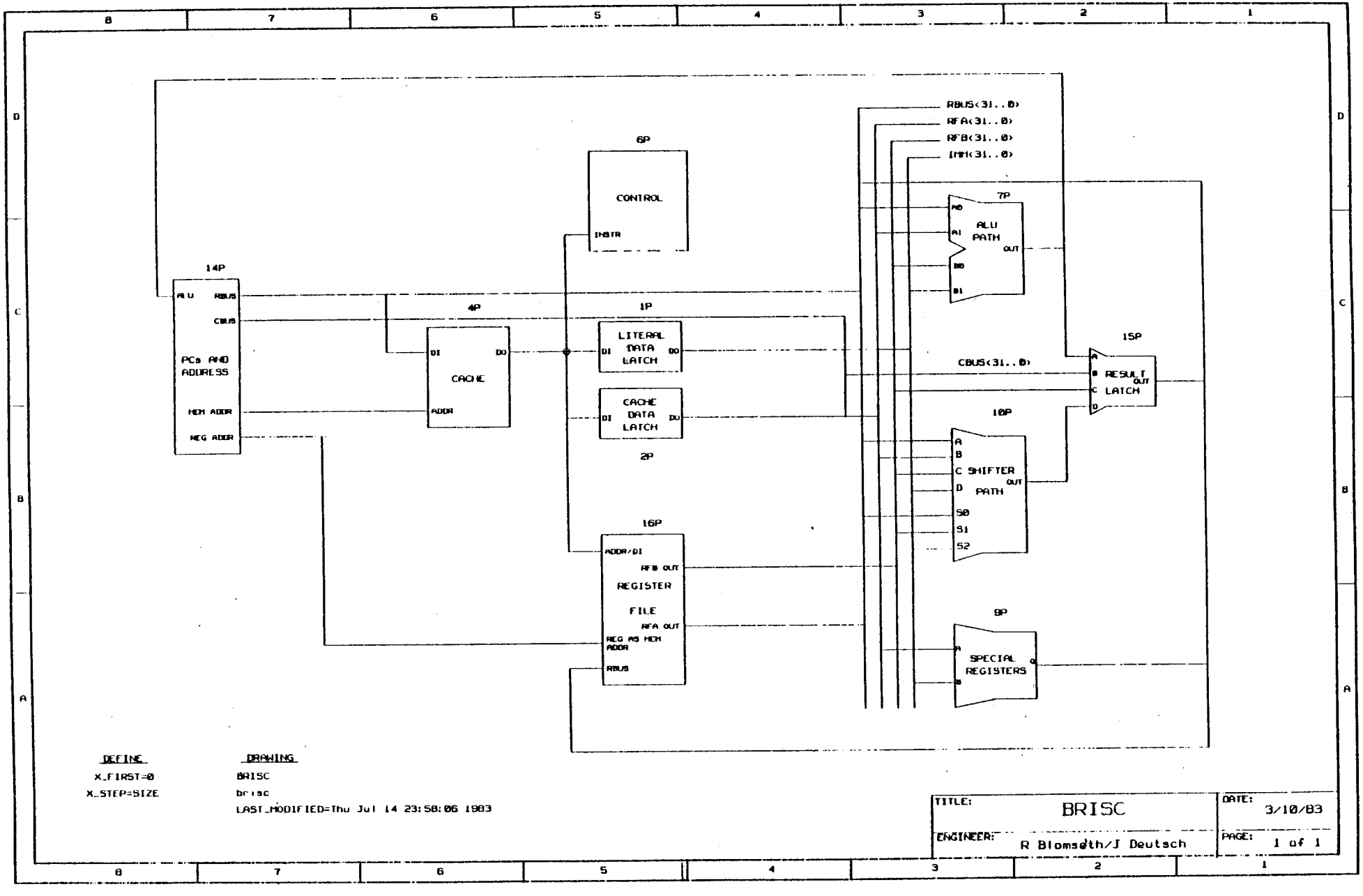
APPENDIX D. BRISC PARTS LIST

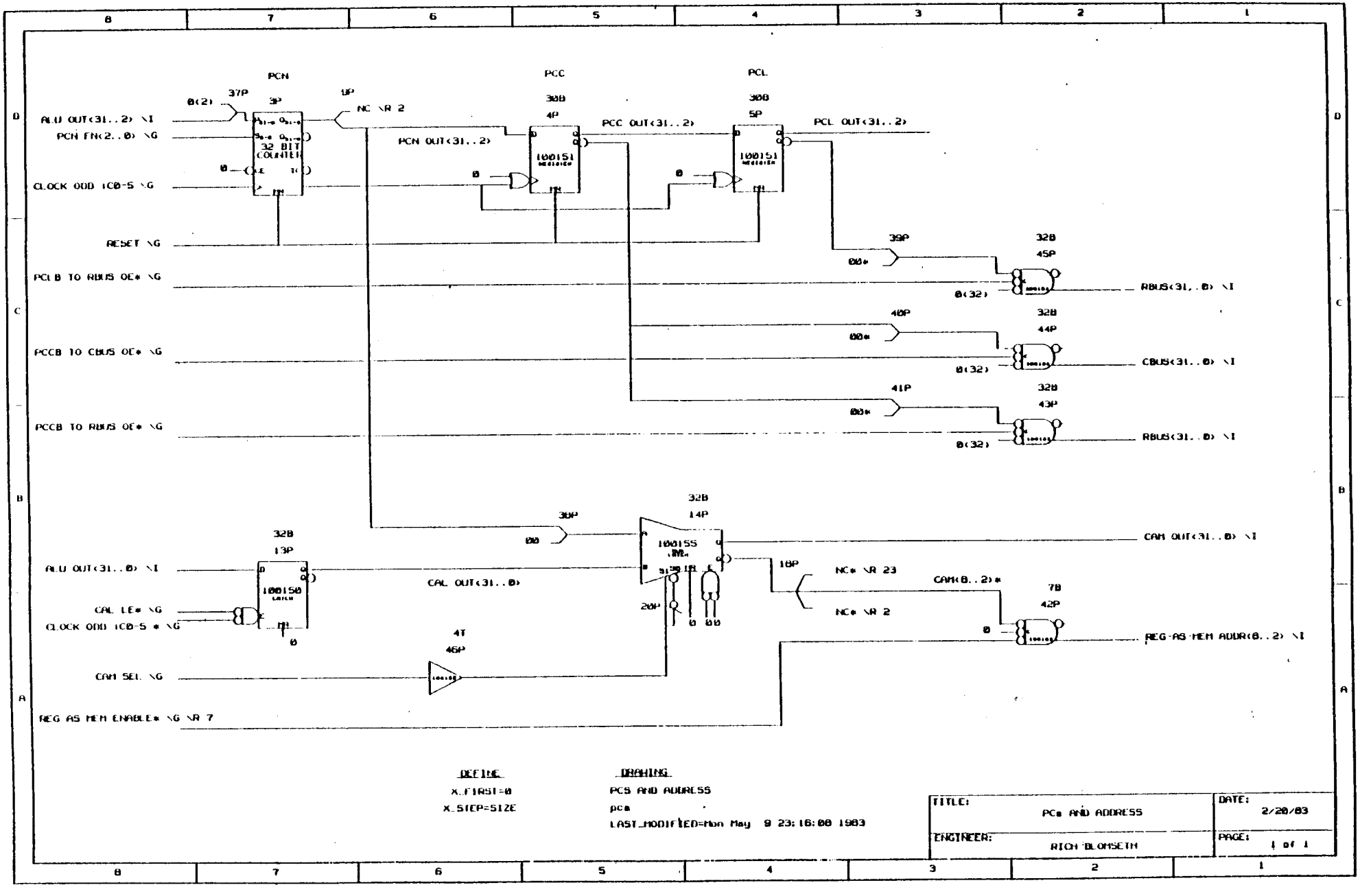
Appendix D
BRISC Parts List

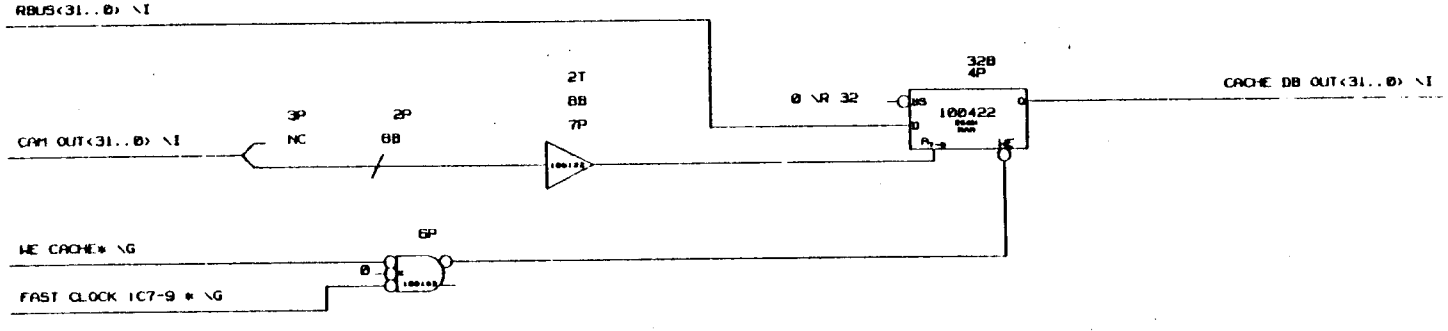
PART NUMBER	QUANTITY	1-100 PRICE		100-1000 PRICE		POWER	
		$ EACH	TOTAL $	$ EACH	TOTAL $	mA EACH	TOTAL mA
100101	6	5.08	30.48	3.61	21.66	38	228
100102	86	5.08	436.88	3.61	310.46	80	6880
100107	3	7.07	21.21	5.02	15.06	96	288
100122	13	5.58	72.54	3.96	51.48	96	1248
100136	16	24.06	384.96	17.07	273.12	283	4528
100141	4	12.15	48.60	8.62	34.48	238	952
100150	47	12.15	571.05	8.62	405.14	159	7473
100151	10	12.90	129.00	9.15	91.50	210	2100
100155	58	15.50	899.00	11.00	638.00	133	7714
100158	16	19.34	309.44	13.73	219.68	205	3280
100171	25	12.65	316.25	8.98	224.50	114	2850
100179	1	15.50	15.50	11.00	11.00	220	220
100181	8	25.54	204.32	18.13	145.04	300	2400
100422	39	32.24	1257.36	22.80	889.20	200	7800
TOTAL	332		$4696.59		$3330.32		49,753

Note: Prices quoted July 1983 by Hamilton-Avnet for Fairchild parts. Power figures are from the Fairchild *P100K ECL DATA Book* except for the 100422, which uses the Fujitsu 100422A-7 power figure.

APPENDIX E. BRISC DRAWINGS


RBUS<31..0>
RFA<31..0>
RFB<31..0>
IMM<31..0>
6P
CONTROL
INSTR
7P
ALU PATH
A0
A1
B0
B1
OUT
14P
ALU
RBUS
CBUS
PCs AND ADDRESS
MEM ADDR
REG ADDR
4P
CACHE
DI
DO
ADDR
1P
LITERAL DATA LATCH
CACHE DATA LATCH
2P
15P
RESULT LATCH
CBUS<31..0>
10P
C SHIFTER PATH
S0
S1
S2
16P
ADDR/DI
RFB OUT
REGISTER FILE
RFA OUT
REG AS MEM ADDR
RBUS
8P
SPECIAL REGISTERS
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
BRISC
brisc
LAST_MODIFIED=Thu Jul 14 23:58:06 1983
TITLE: BRISC
DATE: 3/10/83
ENGINEER: R Blomsath/J Deutsch
PAGE: 1 of 1


PCN
PCC
PCL
ALU OUT<31..2> \I
PCN FN<2..0> \G
CLOCK ODD (C0-5 \G
RESET \G
PCLB TO RBUS OE* \G
PCCB TO CBUS OE* \G
PCCB TO RBUS OE* \G
32 BIT COUNTER
PCN OUT<31..2>
PCC OUT<31..2>
PCL OUT<31..2>
NC \R 2
RBUS<31..0> \I
CBUS<31..0> \I
ALU OUT<31..0> \I
CAL LE* \G
CLOCK ODD (C0-5 * \G
CAL OUT<31..0>
CAM OUT<31..0> \I
CAM SEL \G
NC* \R 23
CAM<8..2>*
REG-AS-MEM ADDR<8..2> \I
REG AS MEM ENABLE* \G \R 7
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
PCS AND ADDRESS
pcs
LAST_MODIFIED=Mon May 9 23:16:00 1983
TITLE:
PCs AND ADDRESS
DATE:
2/20/83
ENGINEER:
RICH BLOMSETH
PAGE:
1 of 1



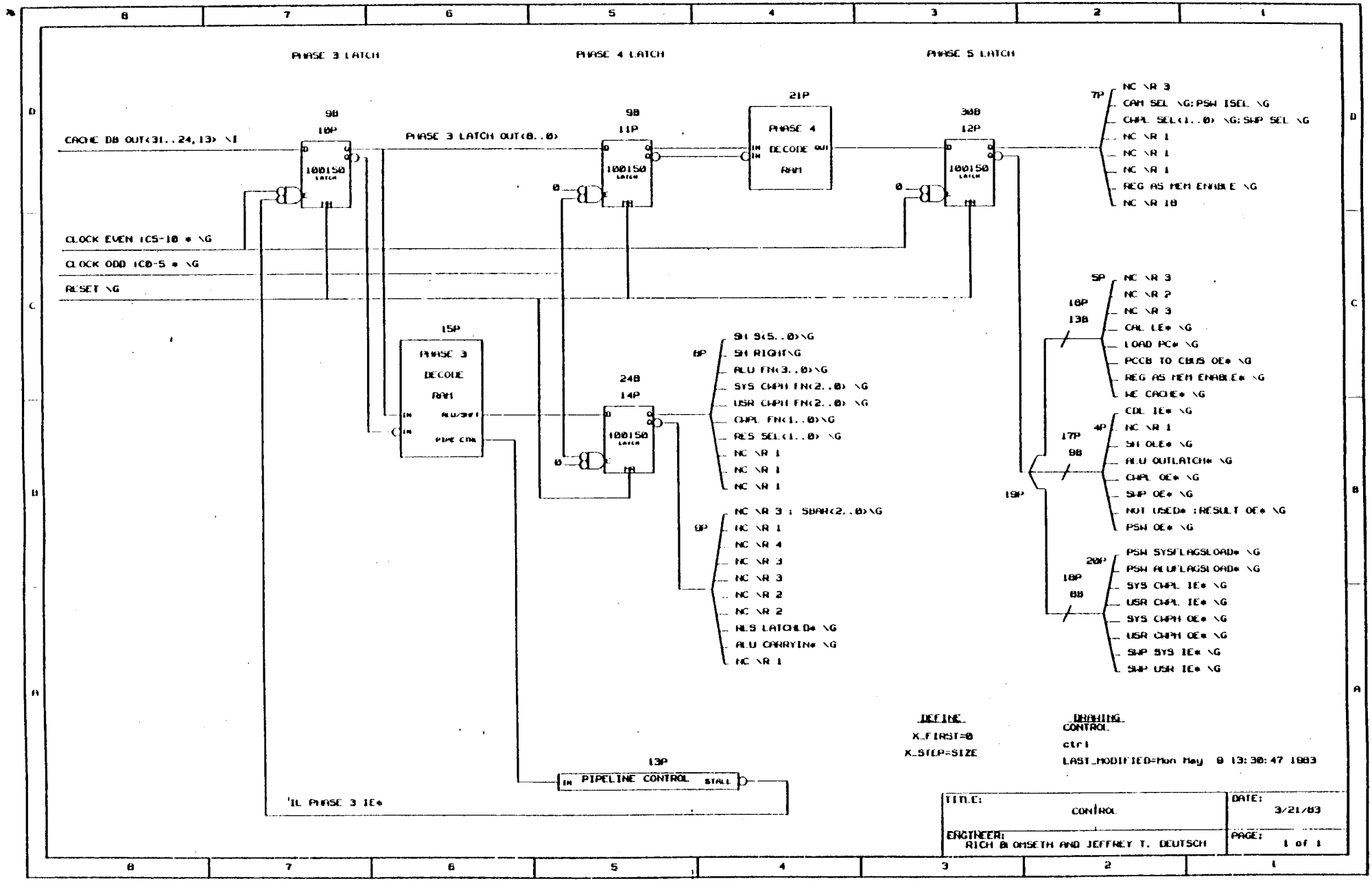
DEFINE
X_FIRST=0
X_STEP=SIZE

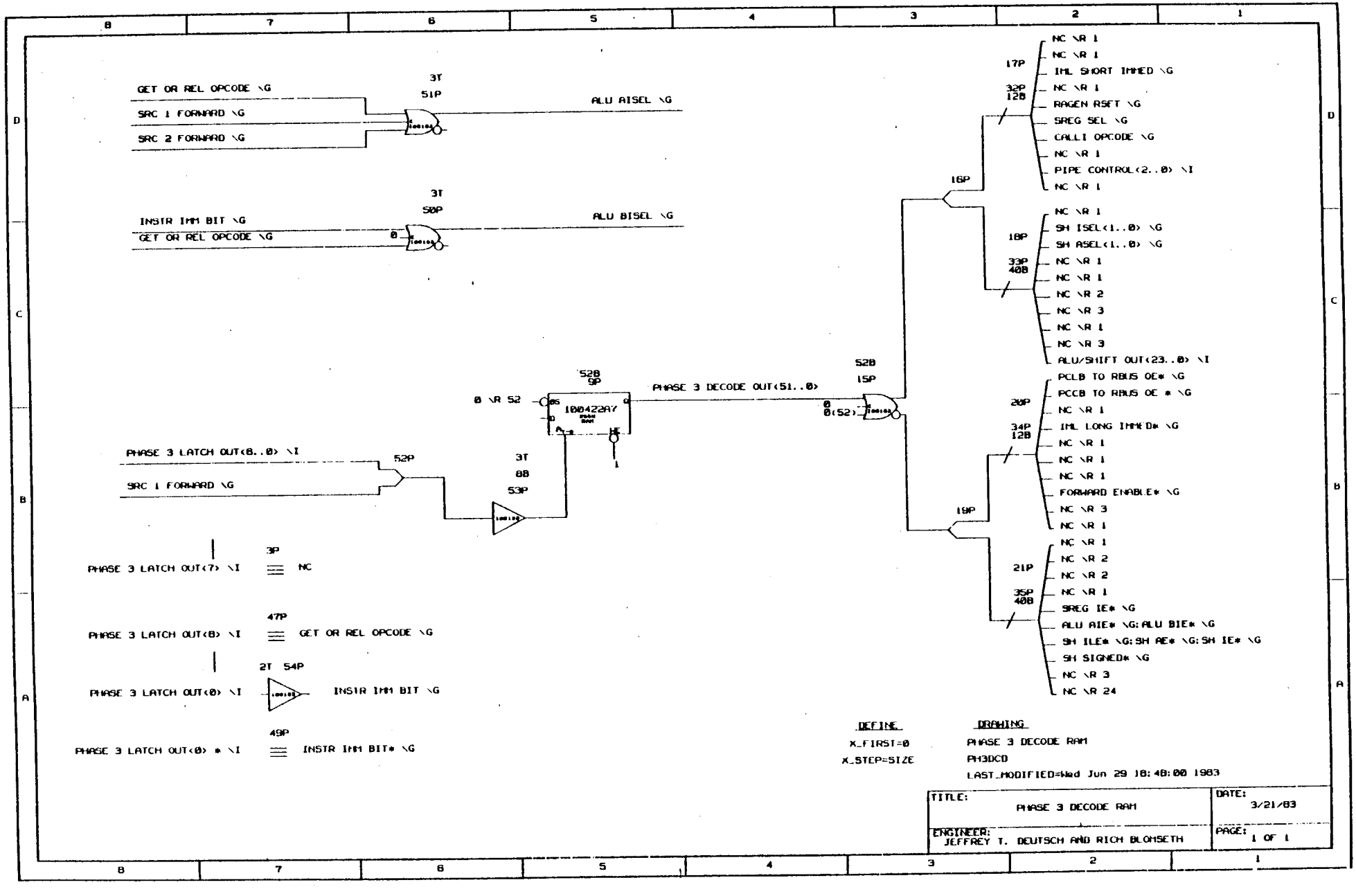
DRAWING
CACHE
cash
LAST_MODIFIED=Mon May 9 13:37:03 1983

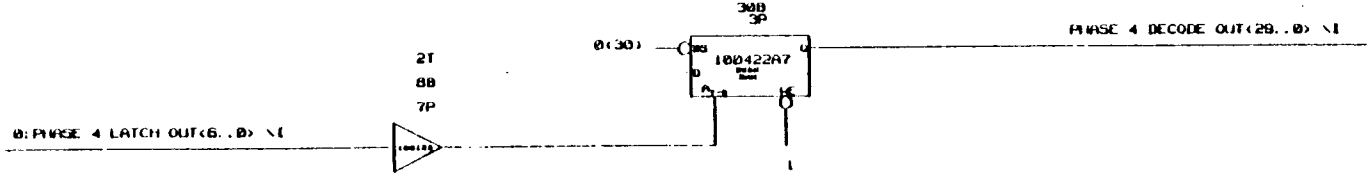
THIS DRAWING IS USED ONLY FOR SIMULATION

IT IS NOT THE ACTUAL CACHE

TITLE: CACHE	DATE: 2/27/83
ENGINEER: RICH BLOMSETH	PAGE: 1 of 1


PHASE 3 LATCH
PHASE 4 LATCH
PHASE 5 LATCH
CACHE DB OUT<31..24,13> \I
PHASE 3 LATCH OUT<8..0>
CLOCK EVEN (C5-10 * \G
CLOCK ODD (C0-5 * \G
RESET \G
9B
10P
100150
LATCH
9B
11P
21P
PHASE 4
DECODE
RAM
30B
12P
7P
NC \R 3
CAM SEL \G;PSW ISEL \G
CHPL SEL<1..0> \G;SHP SEL \G
NC \R 1
NC \R 1
NC \R 1
REG AS MEM ENABLE \G
NC \R 18
15P
PHASE 3
DECODE
RAM
24B
14P
8P
SH SA<5..0>\G
SH RIGHT\G
ALU FN<3..0>\G
SYS CHPH FN<2..0> \G
USR CHPH FN<2..0> \G
CHPL FN<1..0>\G
RES SEL<1..0> \G
NC \R 1
NC \R 1
NC \R 1
9P
NC \R 3 ; SBAR<2..0>\G
NC \R 1
NC \R 4
NC \R 3
NC \R 3
NC \R 2
NC \R 2
ALS LATCHLD* \G
ALU CARRYIN* \G
NC \R 1
5P
18P
13B
NC \R 3
NC \R 2
NC \R 3
CAL LE* \G
LOAD PC* \G
PCCB TO CBUS OE* \G
REG AS MEM ENABLE* \G
WE CACHE* \G
19P
17P
9B
4P
CDL IE* \G
NC \R 1
SH OLE* \G
ALU OUTLATCH* \G
CHPL OE* \G
SHP OE* \G
NOT USED* ;RESULT OE* \G
PSW OE* \G
18P
8B
20P
PSW SYSFLAGSLOAD* \G
PSW ALUFLAGSLOAD* \G
SYS CHPL IE* \G
USR CHPL IE* \G
SYS CHPH OE* \G
USR CHPH OE* \G
SHP SYS IE* \G
SHP USR IE* \G
13P
IN PIPELINE CONTROL STALL
'IL PHASE 3 IE*
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
CONTROL
ctrl
LAST_MODIFIED=Mon May 9 13:30:47 1983
TITLE:
CONTROL
DATE:
3/21/83
ENGINEER:
RICH BLOMSETH AND JEFFREY T. DEUTSCH
PAGE:
1 of 1

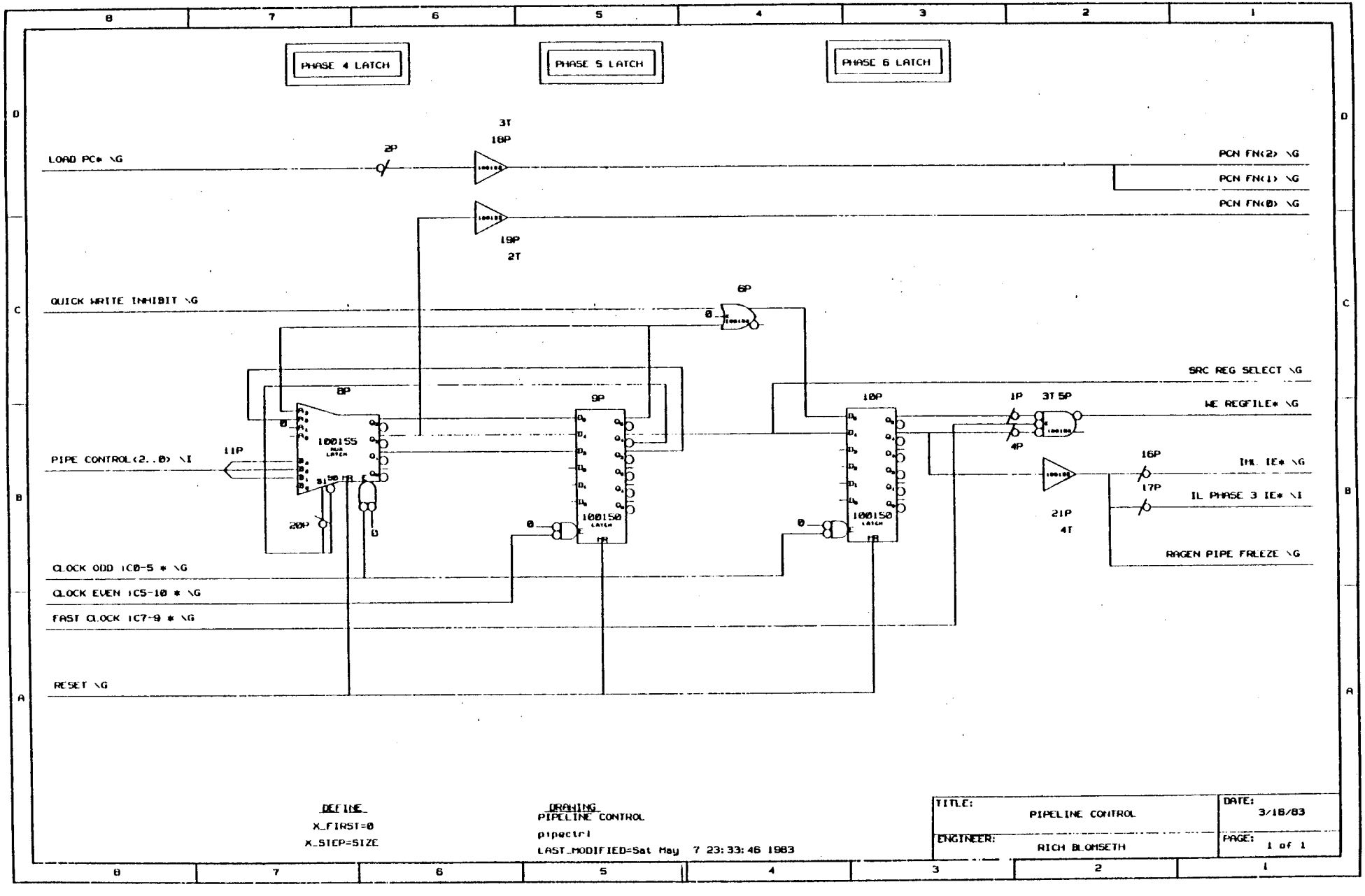

GET OR REL OPCODE \G
SRC 1 FORWARD \G
SRC 2 FORWARD \G
3T
51P
ALU AISEL \G
INSTR IMM BIT \G
GET OR REL OPCODE \G
3T
50P
ALU BISEL \G
0
0 \R 52
52B
9P
10042247
PHASE 3 DECODE OUT(51..0)
52B
15P
0
0(52)
PHASE 3 LATCH OUT(8..0) \I
SRC 1 FORWARD \G
52P
3T
8B
53P
PHASE 3 LATCH OUT(7) \I
3P
NC
PHASE 3 LATCH OUT(8) \I
47P
GET OR REL OPCODE \G
PHASE 3 LATCH OUT(0) \I
2T 54P
INSTR IMM BIT \G
PHASE 3 LATCH OUT(0) * \I
49P
INSTR IMM BIT* \G
16P
17P
32P
12B
NC \R 1
NC \R 1
IML SHORT IMMED \G
NC \R 1
RAGEN RSET \G
SREG SEL \G
CALLI OPCODE \G
NC \R 1
PIPE CONTROL(2..0) \I
NC \R 1
18P
33P
40B
NC \R 1
SH ISEL(1..0) \G
SH ASEL(1..0) \G
NC \R 1
NC \R 1
NC \R 2
NC \R 3
NC \R 1
NC \R 3
ALU/SHIFT OUT(23..0) \I
19P
20P
34P
12B
PCLB TO RBUS OE* \G
PCCB TO RBUS OE * \G
NC \R 1
IML LONG IMMED* \G
NC \R 1
NC \R 1
NC \R 1
FORWARD ENABLE* \G
NC \R 3
NC \R 1
21P
35P
40B
NC \R 1
NC \R 2
NC \R 2
NC \R 1
SREG IE* \G
ALU AIE* \G;ALU BIE* \G
SH ILE* \G;SH AE* \G;SH IE* \G
SH SIGNED* \G
NC \R 3
NC \R 24
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
PHASE 3 DECODE RAM
PH3DCD
LAST_MODIFIED=Wed Jun 29 18:48:00 1983
TITLE:
PHASE 3 DECODE RAM
DATE:
3/21/83
ENGINEER:
JEFFREY T. DEUTSCH AND RICH BLOMSETH
PAGE:
1 OF 1

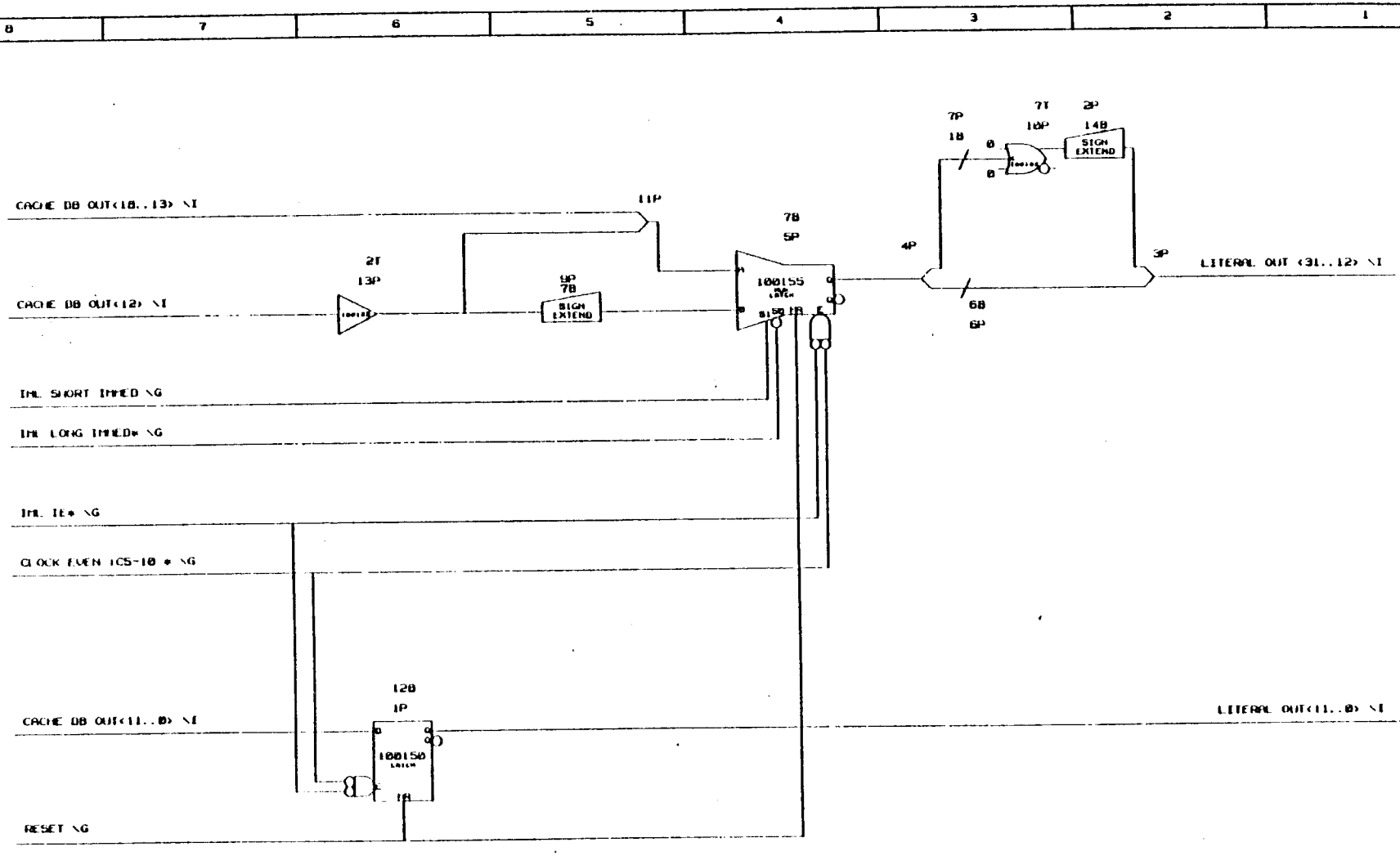


DEFINE
X_FIRST=0
X_STEP=SIZE

DRAWING
PHASE 4 DECODE RAM
ph4dcd
LAST_MODIFIED=Wed Jun 29 18:47:48 1983

TITLE: PHASE 4 DECODE RAM	DATE: 3/22/83
ENGINEER: RICH BLOMSETH	PAGE: 1 of 1

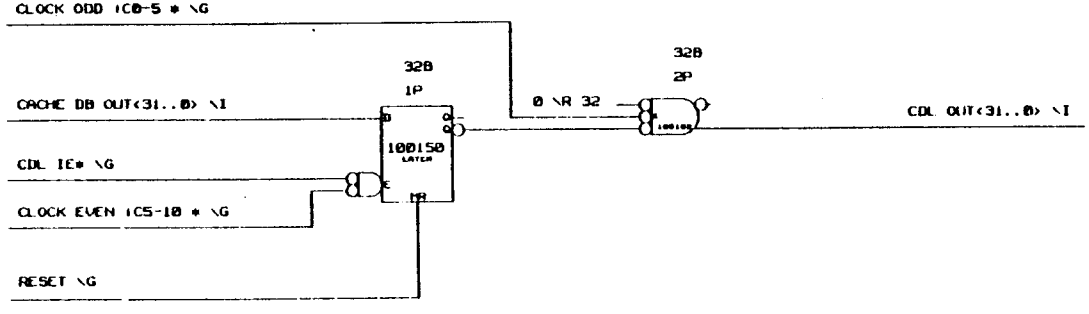

PHASE 4 LATCH
PHASE 5 LATCH
PHASE 6 LATCH
LOAD PC* \G
2P
3T
18P
PCN FN(2) \G
PCN FN(1) \G
PCN FN(0) \G
19P
2T
QUICK WRITE INHIBIT \G
6P
SRC REG SELECT \G
8P
9P
18P
1P
3T 5P
WE REGFILE* \G
PIPE CONTROL(2..0) \I
11P
100155
100150
4P
16P
INH. IE* \G
17P
IL PHASE 3 IE* \I
21P
4T
20P
RAGEN PIPE FREEZE \G
CLOCK ODD (C0-5 *) \G
CLOCK EVEN (C5-10 *) \G
FAST CLOCK (C7-9 *) \G
RESET \G
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
PIPELINE CONTROL
pipectrl
LAST_MODIFIED=Sat May 7 23:33:46 1983
TITLE:
PIPELINE CONTROL
DATE:
3/18/83
ENGINEER:
RICH BLOMSETH
PAGE:
1 of 1



DEFINE
X_FIRST=0
X_STEP-SIZE

DRAWING
LITERAL DATA LATCH
101
LAST_MODIFIED=Sat May 7 23:54:31 1983

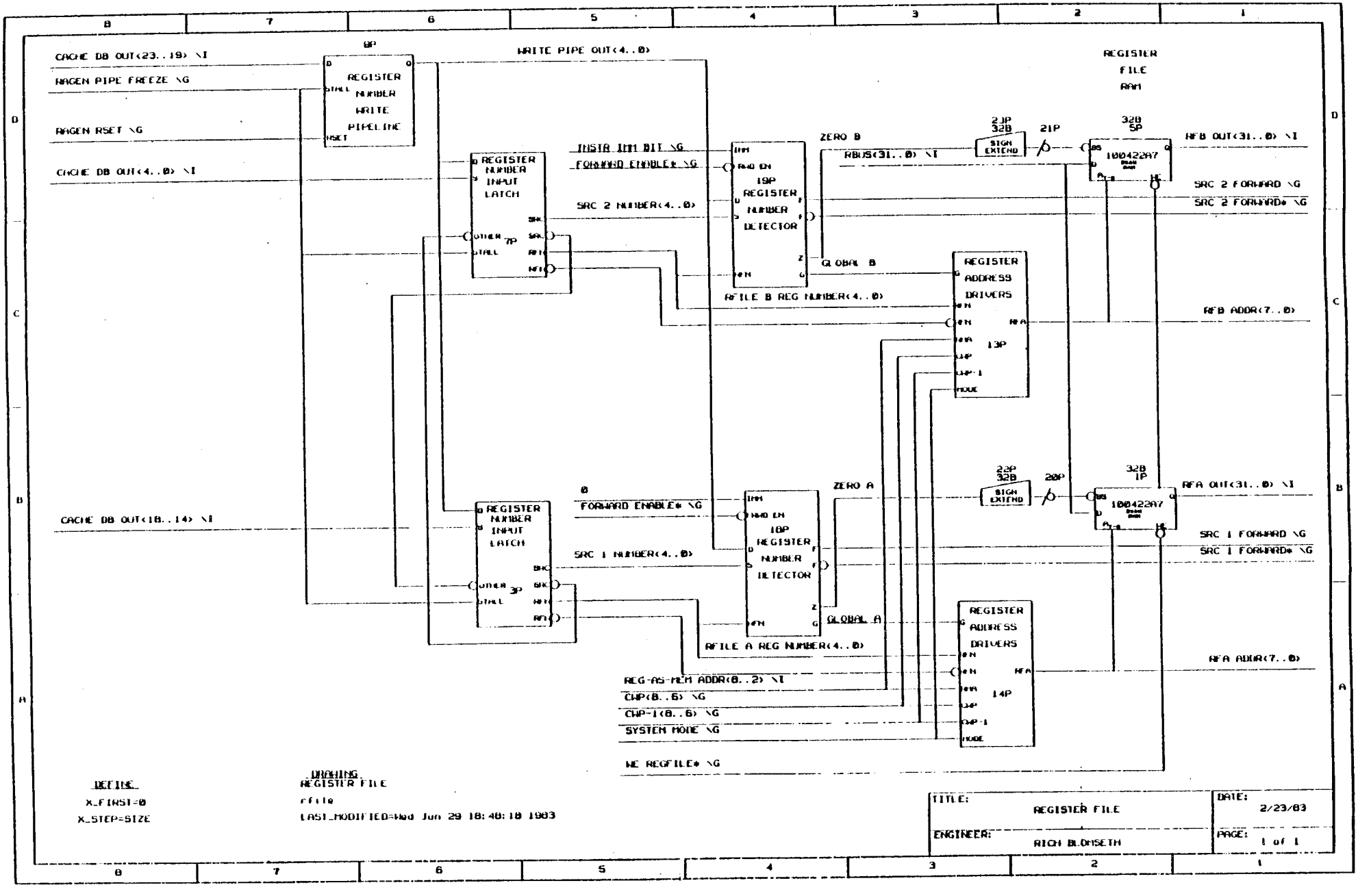
TITLE: LITERAL DATA LATCH	DATE: 2/27/83
ENGINEER: RICH BLOMSETH	PAGE: 1 of 1

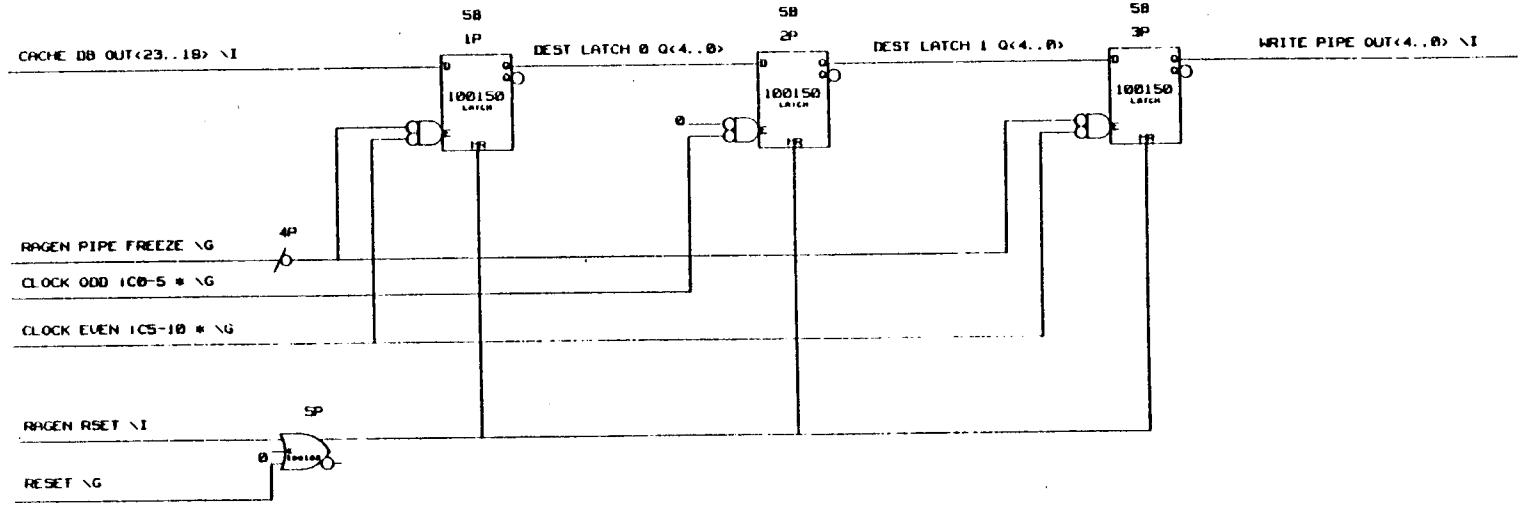


DEFINE
X_FIRST=0
X_STEP=SIZE

DRAWING
CACHE DATA LATCH
cdl
LAST_MODIFIED=Sat May 7 23:57:50 1983

TITLE:	CACHE DATA LATCH	DATE:	2/27/83
ENGINEER:	RICH BLOMSETH	PAGE:	1 of 1


CACHE DB OUT<23..19> \I
RAGEN PIPE FREEZE \G
RAGEN RSET \G
CACHE DB OUT<4..0> \I
CACHE DB OUT<18..14> \I
8P
REGISTER
NUMBER
WRITE
PIPELINE
WRITE PIPE OUT<4..0>
REGISTER NUMBER INPUT LATCH
7P
3P
INSTR IMM BIT \G
FORWARD ENABLE* \G
SRC 2 NUMBER<4..0>
SRC 1 NUMBER<4..0>
19P
18P
REGISTER NUMBER DETECTOR
ZERO B
ZERO A
GLOBAL B
GLOBAL A
RFILE B REG NUMBER<4..0>
RFILE A REG NUMBER<4..0>
REGISTER ADDRESS DRIVERS
13P
14P
REG-AS-MEM ADDR<8..2> \I
CHP<8..6> \G
CHP-1<8..6> \G
SYSTEM MODE \G
WE REGFILE* \G
RBUS<31..0> \I
22P 32B
SIGN EXTEND
21P
20P
32B 5P
32B 1P
100422A7
RFB OUT<31..0> \I
RFA OUT<31..0> \I
SRC 2 FORWARD \G
SRC 2 FORWARD* \G
SRC 1 FORWARD \G
SRC 1 FORWARD* \G
RFB ADDR<7..0>
RFA ADDR<7..0>
REGISTER FILE RAM
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
REGISTER FILE
rfile
LAST_MODIFIED=Wed Jun 29 18:48:18 1983
TITLE: REGISTER FILE
ENGINEER: RICH BLOMSETH
DATE: 2/23/83
PAGE: 1 of 1



DEFINE

X_FIRST=0

X_STEP=SIZE

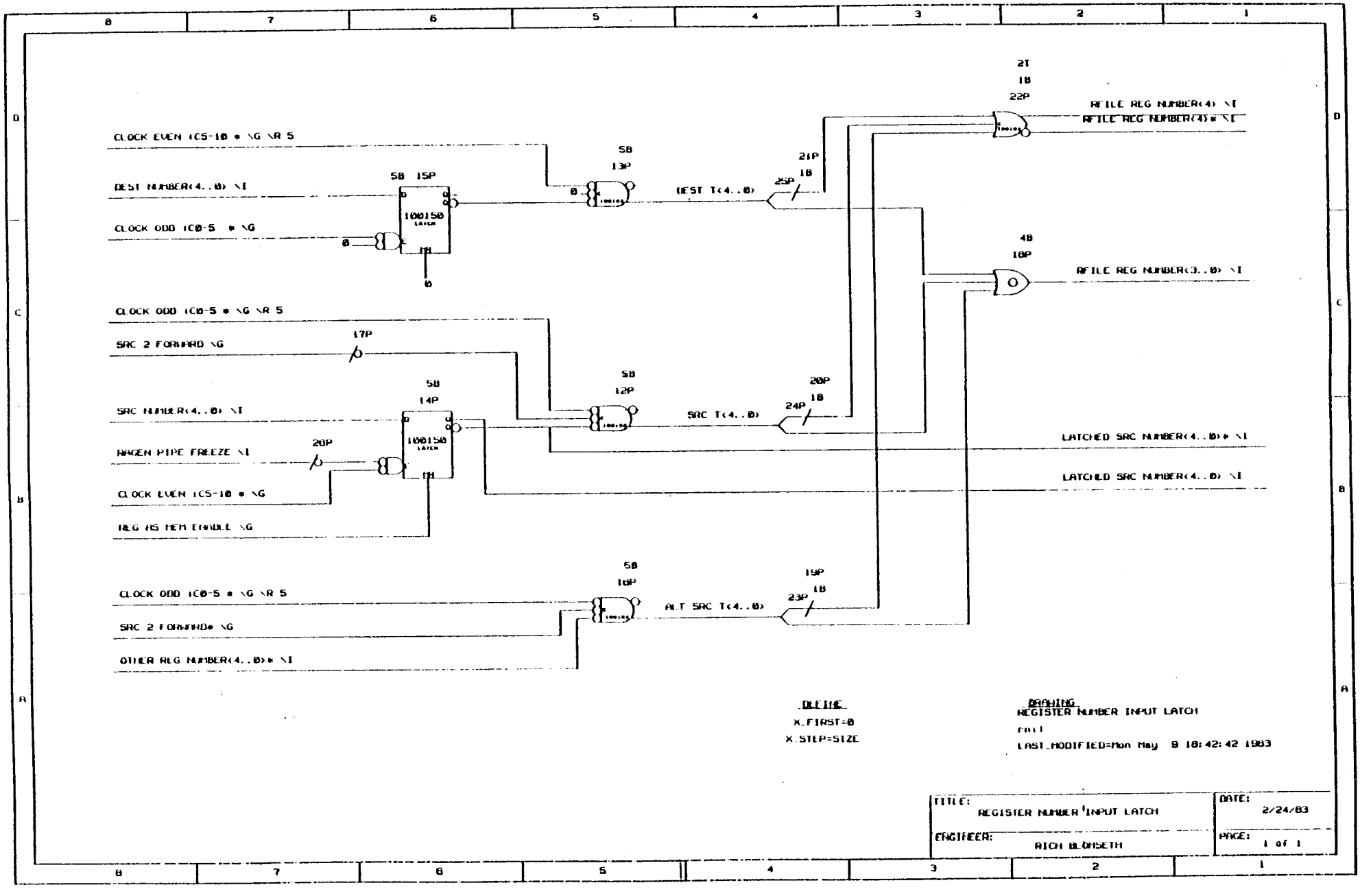
DRAWING

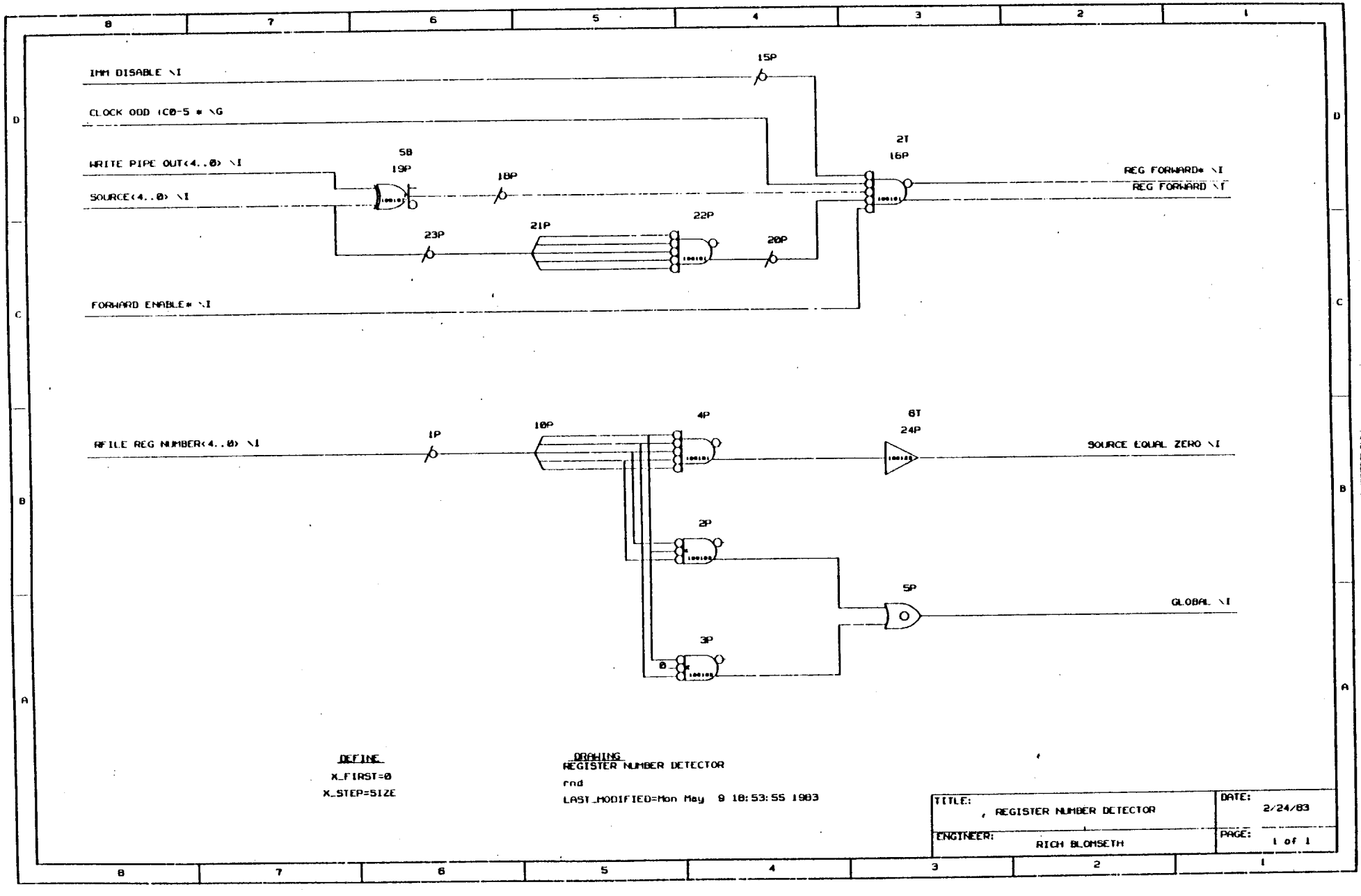
REGISTER NUMBER WRITE PIPELINE

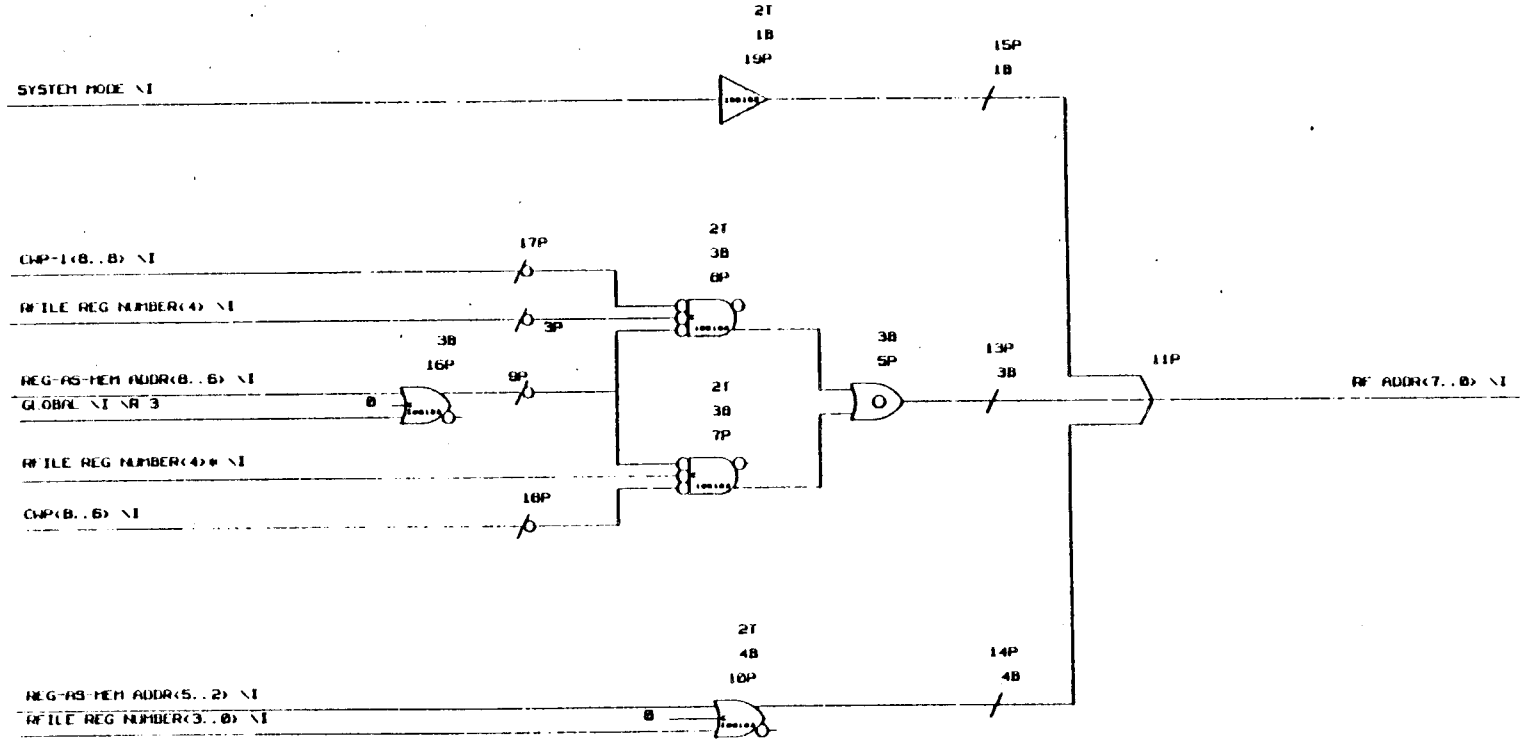
rnwp

LAST_MODIFIED=Sat May 7 23:07:42 1983

TITLE: REGISTER NUMBER WRITE PIPELINE	DATE: 2/23/83
ENGINEER: RICH BLOMSETH	PAGE: 1 of 1


CLOCK EVEN (C5-10 * \G \R 5
DEST NUMBER(4..0) \I
CLOCK ODD (C0-5 * \G
5B 15P
100150
LATCH
5B
13P
DEST T(4..0)
25P
21P
1B
2T
1B
22P
RFILE REG NUMBER(4) \I
RFILE REG NUMBER(4)* \I
4B
10P
RFILE REG NUMBER(3..0) \I
CLOCK ODD (C0-5 * \G \R 5
SRC 2 FORWRD \G
17P
SRC NUMBER(4..0) \I
5B
14P
5B
12P
SRC T(4..0)
24P
20P
1B
RAGEN PIPE FREEZE \I
20P
CLOCK EVEN (C5-10 * \G
REG FIS MEM ENABLE \G
LATCHED SRC NUMBER(4..0)* \I
LATCHED SRC NUMBER(4..0) \I
CLOCK ODD (C0-5 * \G \R 5
SRC 2 FORWRD* \G
OTHER REG NUMBER(4..0)* \I
5B
10P
ALT SRC T(4..0)
23P
19P
1B
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
REGISTER NUMBER INPUT LATCH
FAST
LAST_MODIFIED=Mon May 9 18:42:42 1983
TITLE:
REGISTER NUMBER INPUT LATCH
DATE:
2/24/83
ENGINEER:
RICH BLOMSETH
PAGE:
1 of 1


IMM DISABLE \I
15P
CLOCK ODD (C0-5 * \G
WRITE PIPE OUT<4..0> \I
SOURCE<4..0> \I
5B
19P
18P
2T
16P
REG FORWARD* \I
REG FORWARD \I
23P
21P
22P
20P
FORWARD ENABLE* \I
RFILE REG NUMBER<4..0> \I
1P
10P
4P
6T
24P
SOURCE EQUAL ZERO \I
2P
5P
GLOBAL \I
3P
0
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
REGISTER NUMBER DETECTOR
rnd
LAST_MODIFIED=Mon May 9 10:53:55 1983
TITLE:
REGISTER NUMBER DETECTOR
DATE:
2/24/83
ENGINEER:
RICH BLOMSETH
PAGE:
1 of 1



DEFINE

X_FIRST=0

X_STEP=SIZE

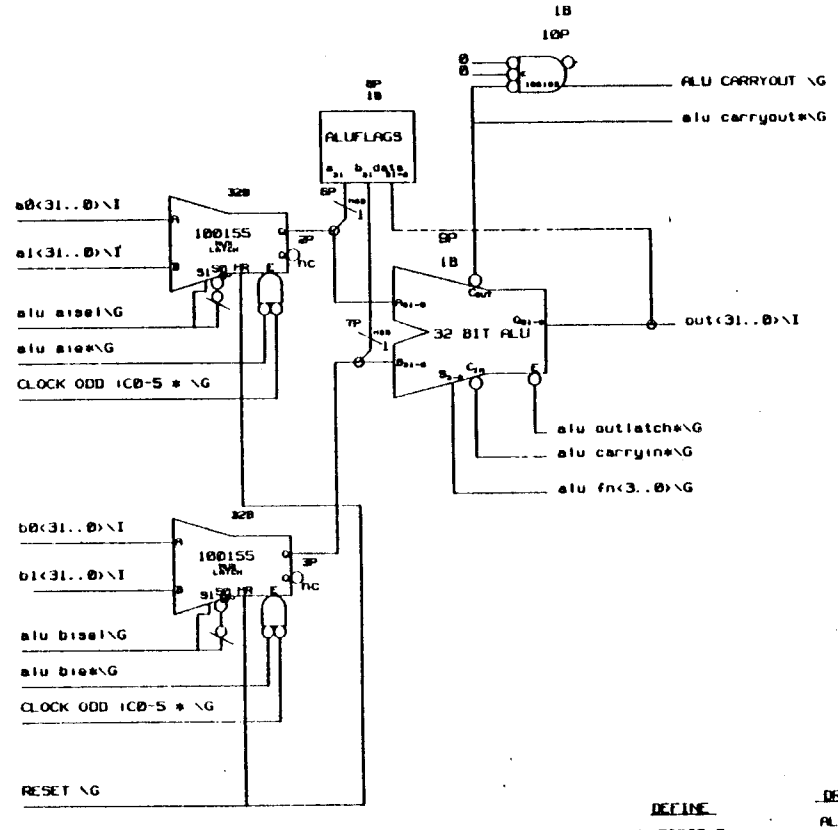
DRAWING

REGISTER ADDRESS DRIVERS

read

LAST_MODIFIED=Mon May 9 10:48:13 1983

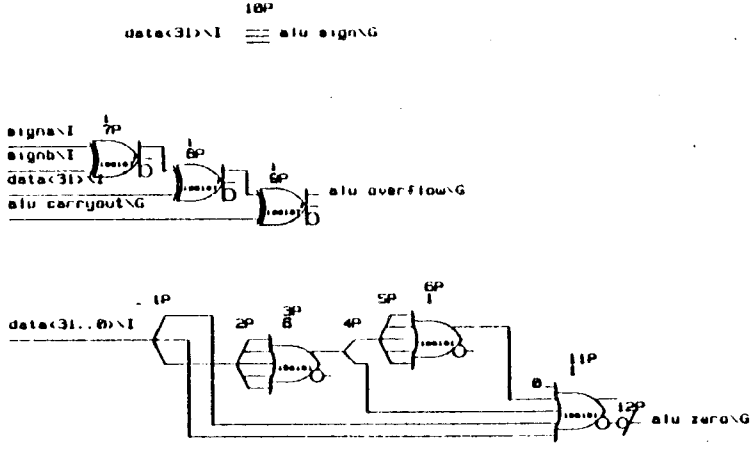
TITLE: REGISTER ADDRESS DRIVERS	DATE: 2/27/83
ENGINEER: RICH BLOMSETH	PAGE: 1 of 1



DEFINE
X_FIRST=0
X_STEP=SIZE

DRAWING
ALUPATH
alupath
LAST_MODIFIED=Fri May 13 16:17:18 1983

TITLE: alupath	DATE: 2/27/83
ENGINEER: Jeffrey T. Deutsch	PAGE: 1 of 1



DEFINE

X_FIRST=0

X_STEP=SIZE

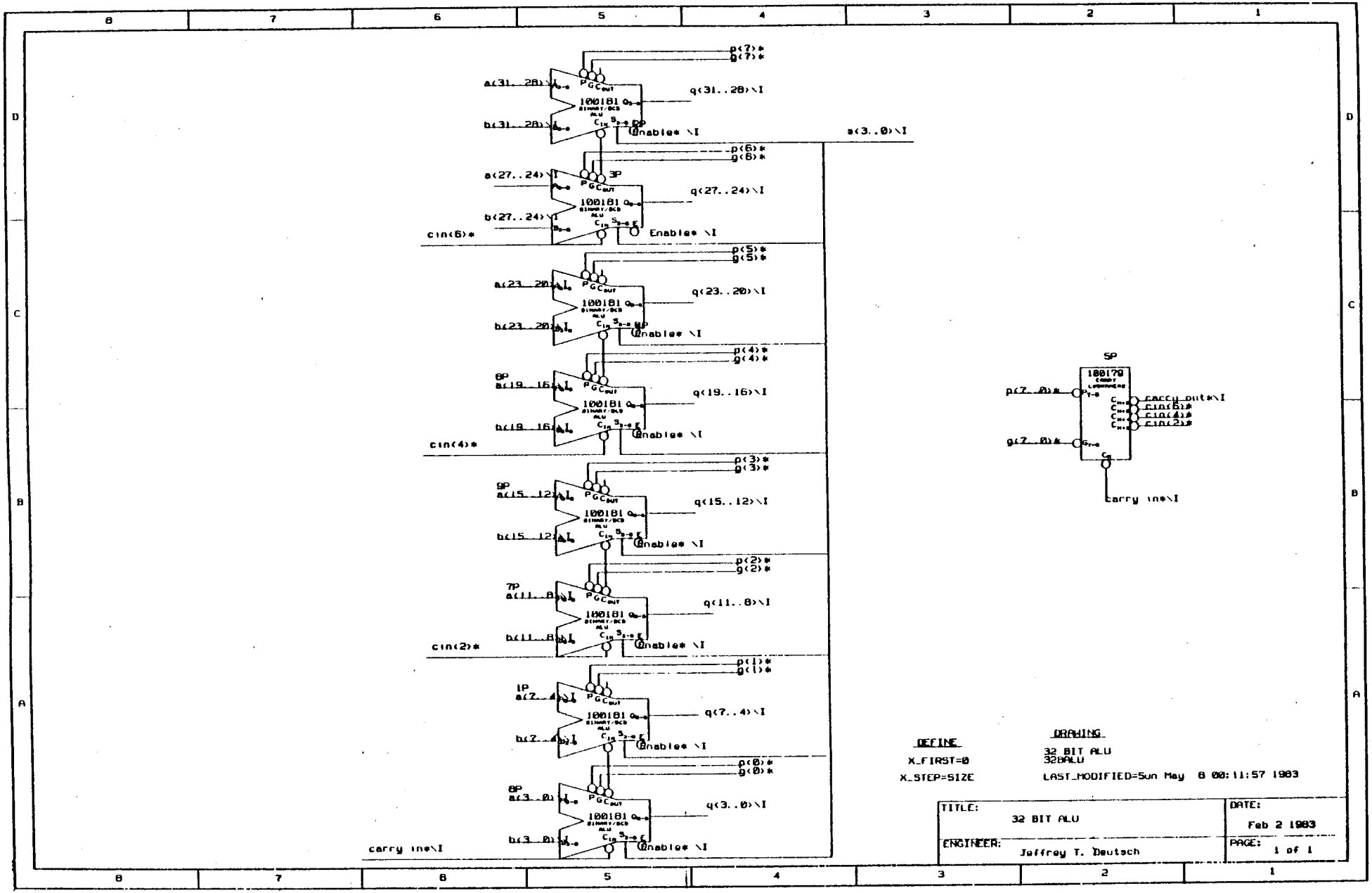
DRAWING

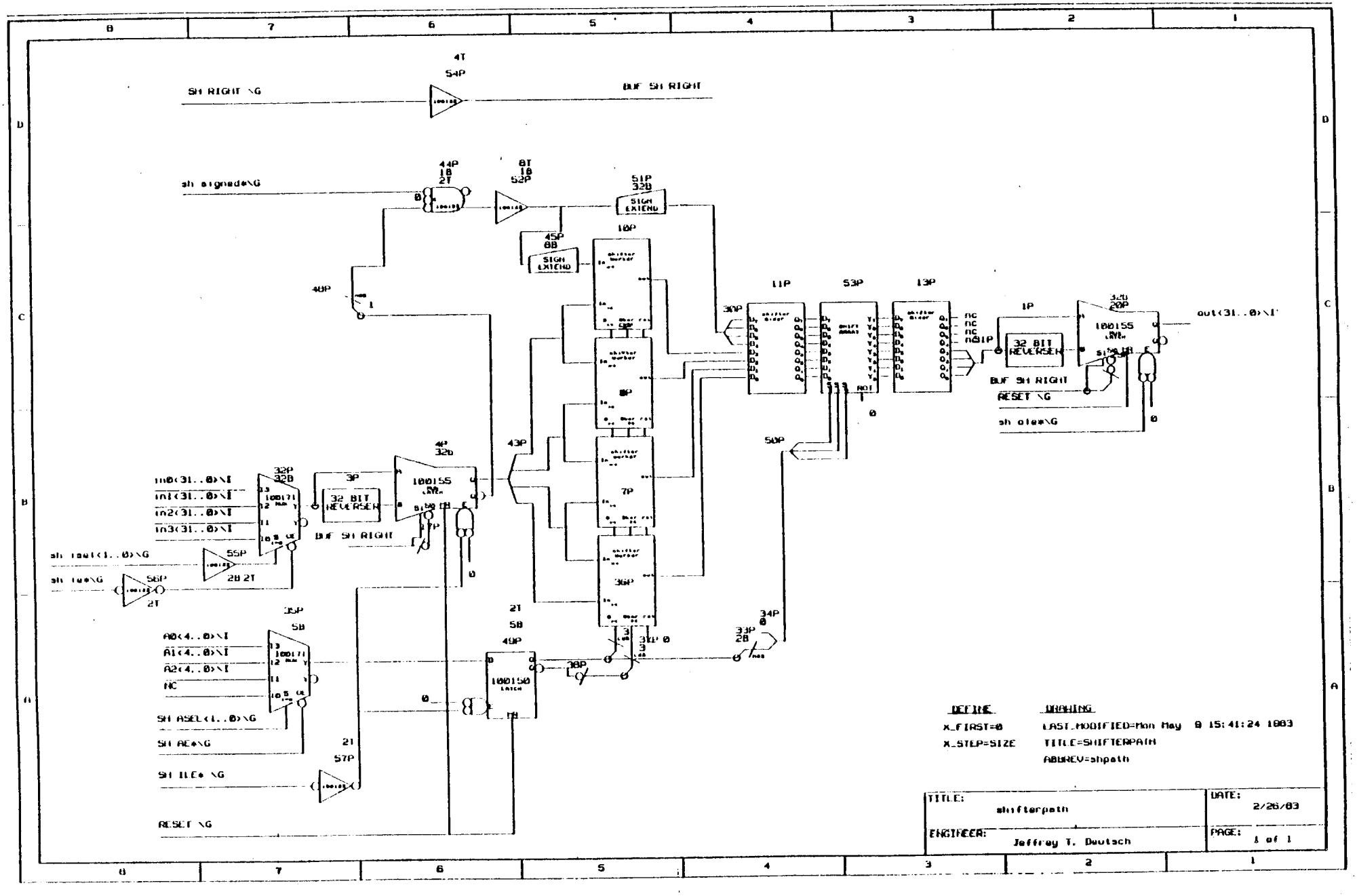
ALUFLAGS

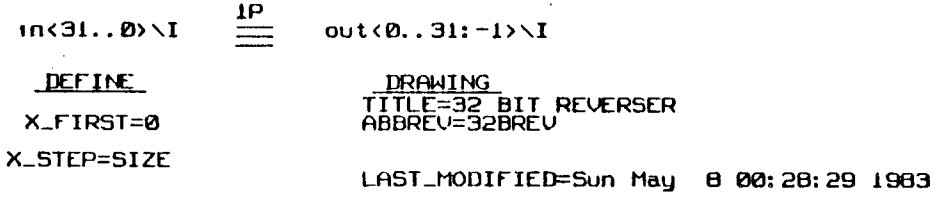
AFLAGS

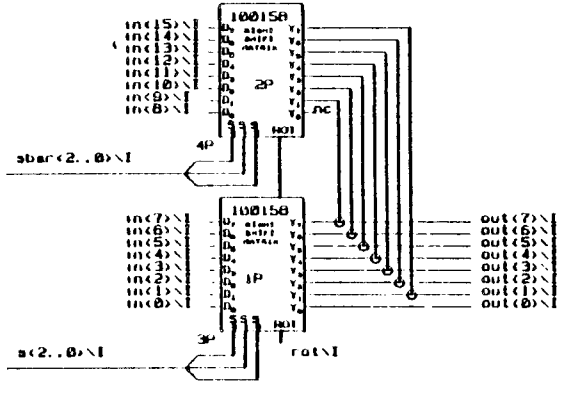
LAST_MODIFIED=Sun May 8 00:00:09 1983

TITLE: aluflags	DATE: 3/17/83
ENGINEER: Jeff Dautch	PAGE: 1 of 1


p(7)*
g(7)*
a(31..28)\I
b(31..28)\I
100181
q(31..28)\I
Enable* \I
s(3..0)\I
p(6)*
g(6)*
3P
a(27..24)\I
b(27..24)\I
q(27..24)\I
cin(6)*
Enable* \I
p(5)*
g(5)*
a(23..20)\I
b(23..20)\I
q(23..20)\I
Enable* \I
p(4)*
g(4)*
8P
a(19..16)\I
b(19..16)\I
q(19..16)\I
cin(4)*
Enable* \I
p(3)*
g(3)*
8P
a(15..12)\I
b(15..12)\I
q(15..12)\I
Enable* \I
p(2)*
g(2)*
7P
a(11..8)\I
b(11..8)\I
q(11..8)\I
cin(2)*
Enable* \I
p(1)*
g(1)*
1P
a(7..4)\I
b(7..4)\I
q(7..4)\I
Enable* \I
p(0)*
g(0)*
8P
a(3..0)\I
b(3..0)\I
q(3..0)\I
carry in*\I
Enable* \I
5P
100170
p(7..0)*
g(7..0)*
carry out*\I
cin(6)*
cin(4)*
cin(2)*
carry in*\I
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
32 BIT ALU
32BALU
LAST_MODIFIED=Sun May 8 00:11:57 1983
TITLE:
32 BIT ALU
DATE:
Feb 2 1983
ENGINEER:
Jeffrey T. Deutsch
PAGE:
1 of 1


SH RIGHT \G
BUF SH RIGHT
sh signed*\G
SIGN EXTEND
32 BIT REVERSER
100155
BUF SH RIGHT
RESET \G
sh oe*\G
out(31..0)\I
in0(31..0)\I
in1(31..0)\I
in2(31..0)\I
in3(31..0)\I
sh sel(1..0)\G
sh oe*\G
A0(4..0)\I
A1(4..0)\I
A2(4..0)\I
NC
SH ASEL(1..0)\G
SH AE*\G
SH ILE* \G
RESET \G
100150
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
LAST_MODIFIED=Mon May 0 15:41:24 1993
TITLE=SHIFTERPATH
ABBREV=shpath
TITLE:
shifterpath
DATE:
2/26/93
ENGINEER:
Jeffrey T. Deutsch
PAGE:
1 of 1


in<31..0>\I
IP
out<0..31:-1>\I
DEFINE
X_FIRST=0
X_STEP=SIZE
DRAWING
TITLE=32 BIT REVERSER
ABBREV=32BREV
LAST_MODIFIED=Sun May 8 00:28:29 1983



DEFINE

X_FIRST=0

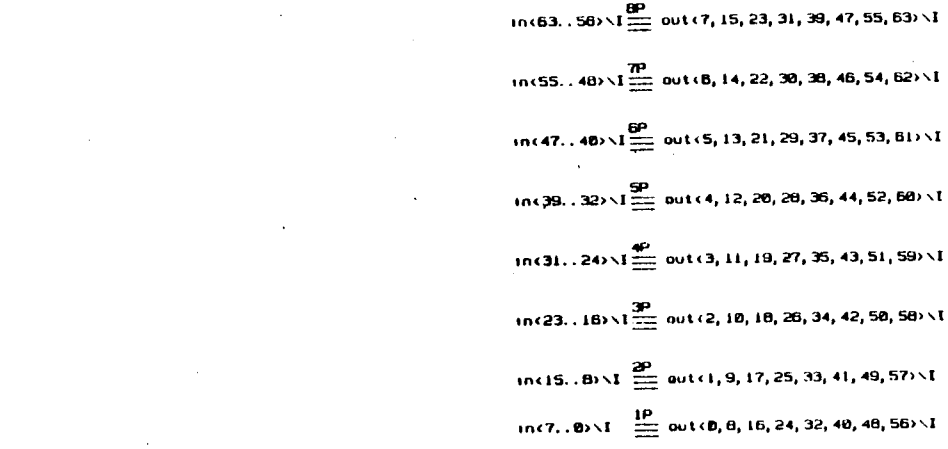
X_STEP=SIZE

DRAWING

SHIFTER WORKER
SHFTWRK

LAST_MODIFIED=Sun May 8 00:34:24 1983

TITLE: shifter worker	DATE: 2/25/83
ENGINEER: Jeffrey T. Deutsch	PAGE: 1 of 1



DEFINE

X_FIRST=0

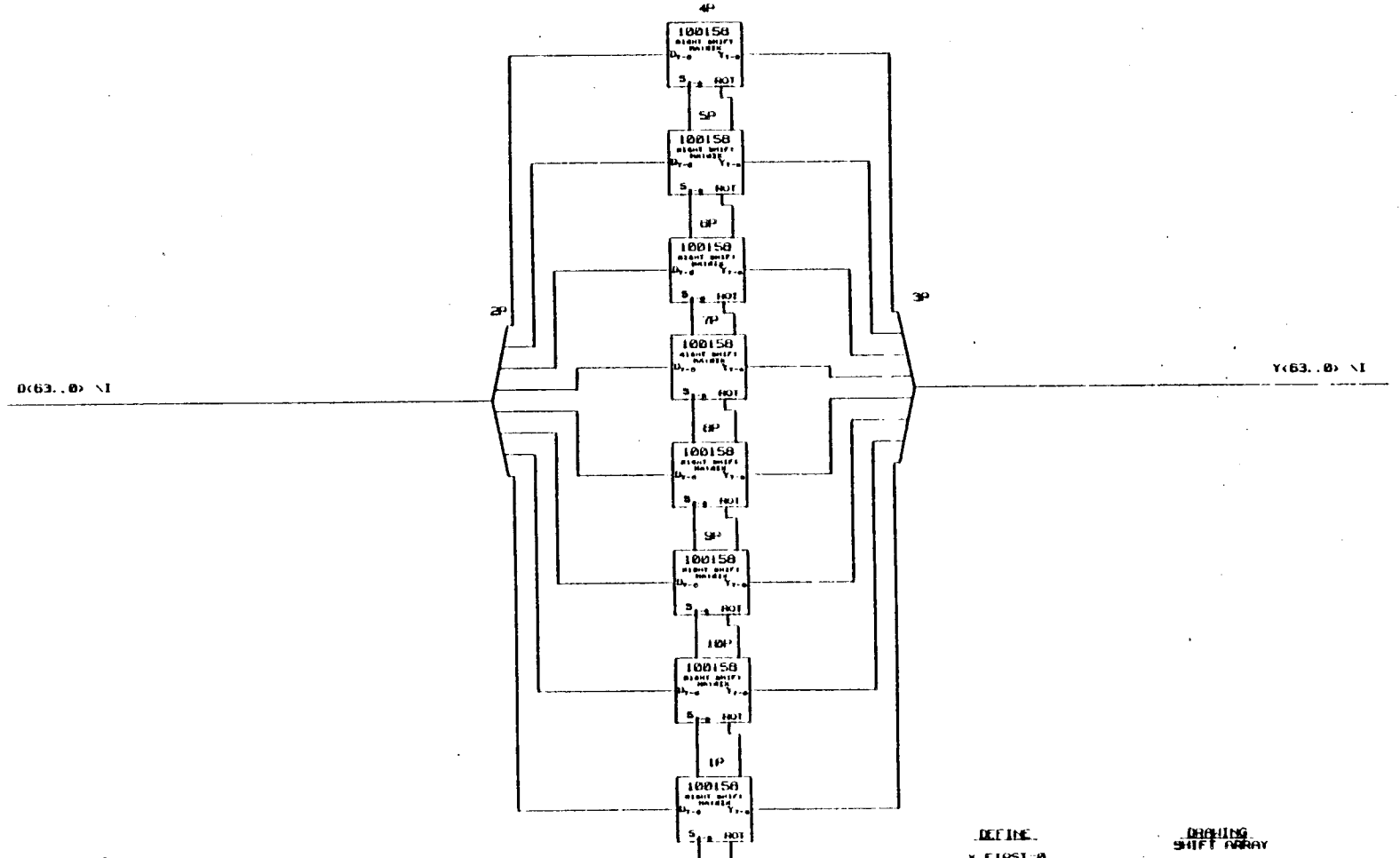
X_STEP=size

DRAWING

SHIFTER MIXER
SHFTMXR

LAST_MODIFIED=Sun May 8 00:35:14 1983

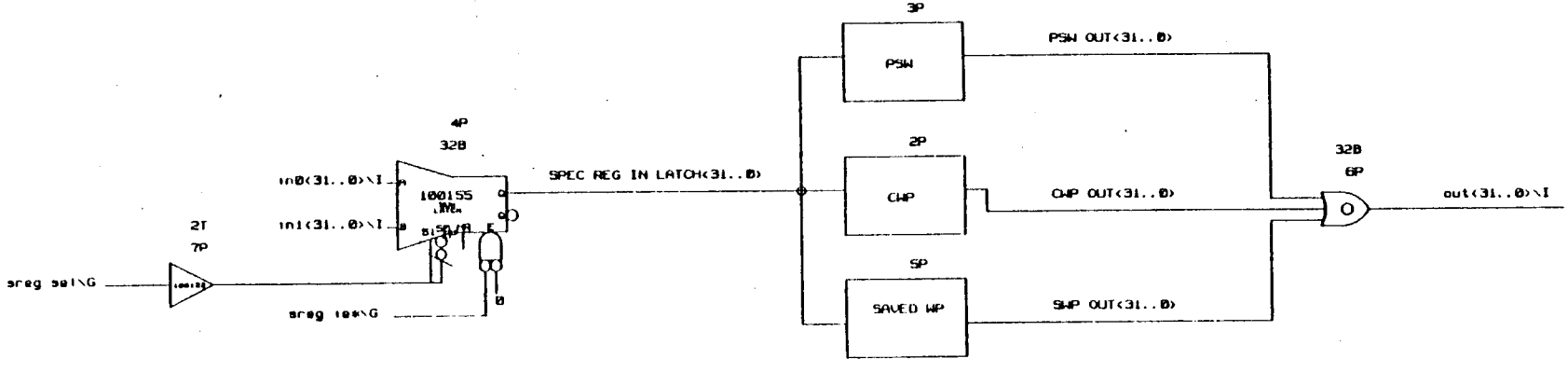
TITLE: shifter mixer	DATE: 2/25/83
ENGINEER: Jeffrey T. Deutsch	PAGE: 1 of 1



DEFINE.
X FIRST=0
X.STEP=SIZE

DRAWING
SHIFT ARRAY
shiftary
LAST_MODIFIED=Mon May 9 11:33:38 1983

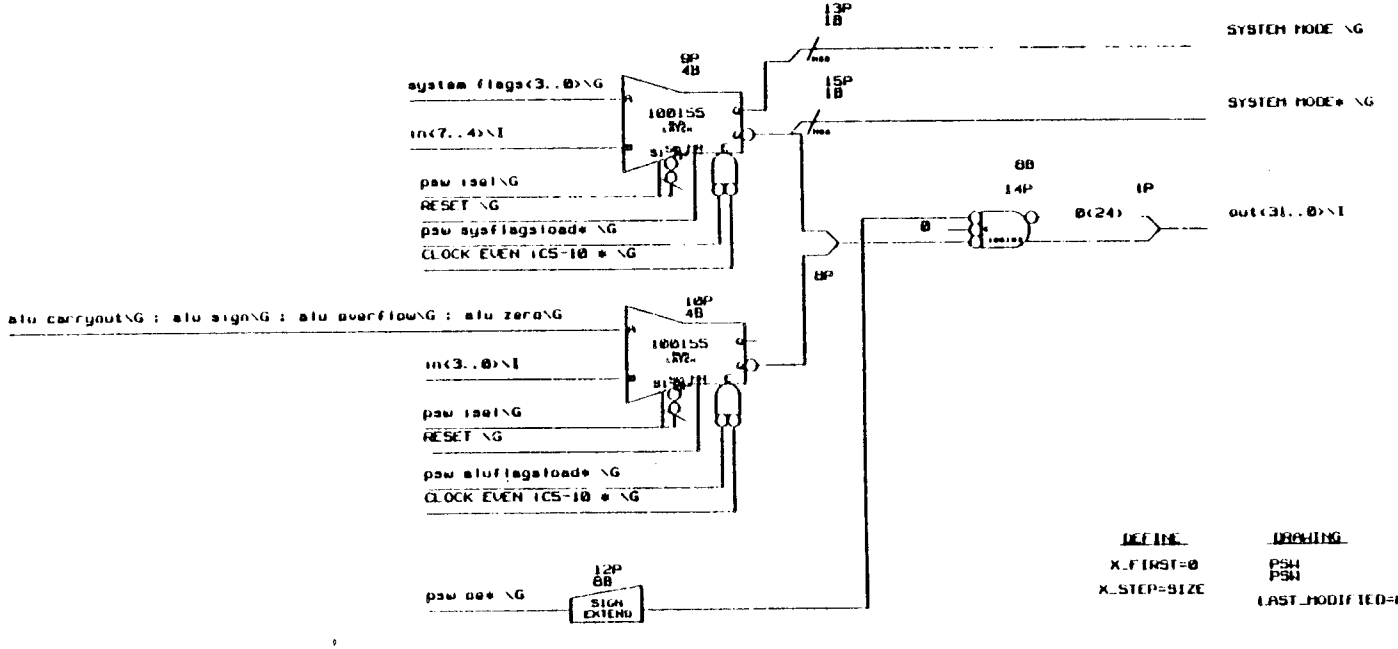
TITLE:	SHIFT ARRAY	DATE:	5/9/83
ENGINEER:	RICH BLOMSETH	PAGE:	1 of 1



DEFINE
X_FIRST=0
X_STEP=SIZE

DRAWING
SPECIAL REGISTERS
SPECREGS
LAST_MODIFIED=Thu May 12 11:45:38 1983

TITLE: special registers	DATE: 3/11/83
ENGINEER: Jeffrey T. Deutsch	PAGE: 1 of 1

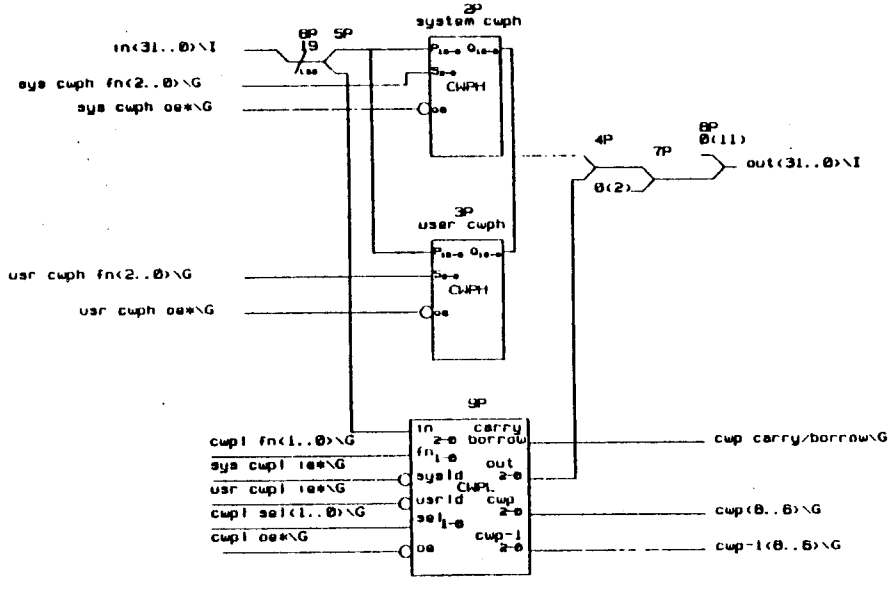


DEFINE
X_FIRST=0
X_STEP=SIZE

DRAWING
PSH
PSH

LAST_MODIFIED=Mon May 9 16:15:34 1983

TITLE: psw	DATE: 3/11/83
ENGINEER: Jeffrey T. Deutsch	PAGE: 1 of 1



DEFINE

X_FIRST=0

X_STEP=SIZE

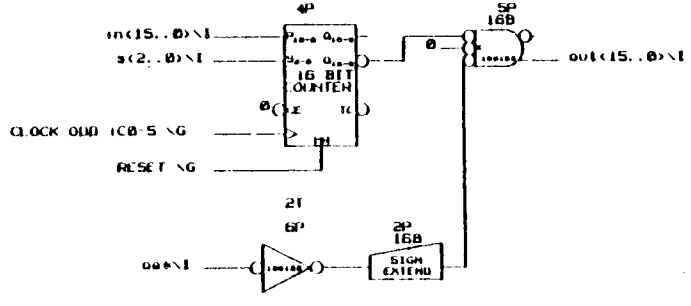
DRAWING

CWP

CWP

LAST_MODIFIED=Sun May 8 00:38:05 1983

TITLE: Current Window Pointer	DATE: 3/15/83
ENGINEER: Jeffrey T. Deutsch	PAGE: 1 of 1

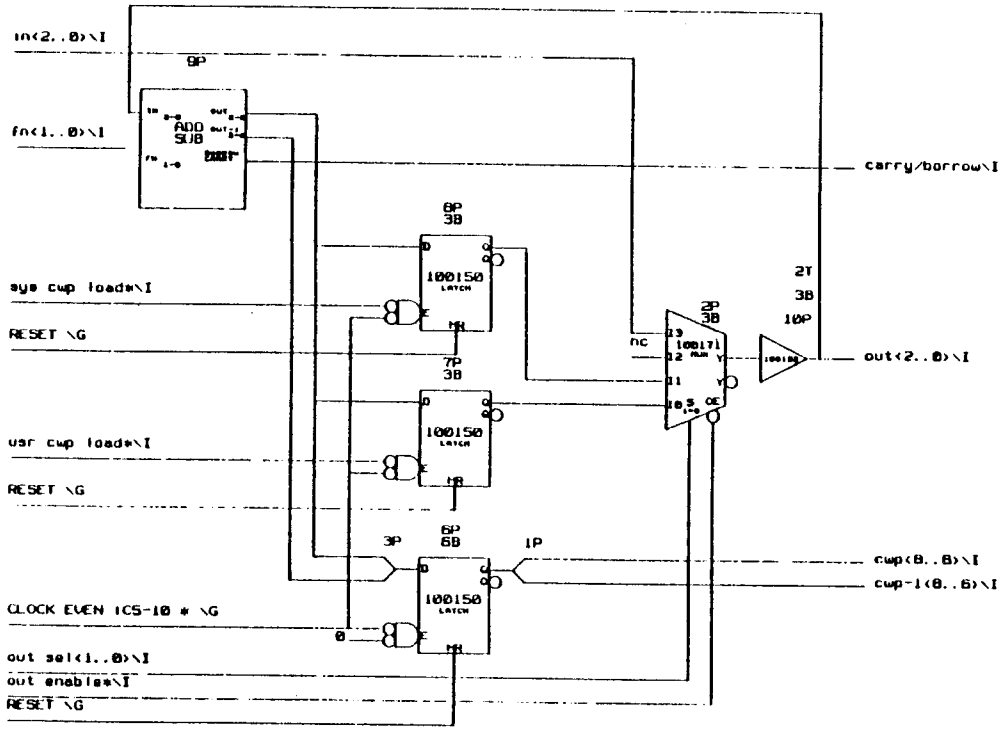


DEFINE
X_FIRST=0
X_STEP=SIZE

DRAWING
CWPH
cuph

LAST_MODIFIED=Mon May 9 16:14:35 1983

TITLE: CWPH	DATE: 3/15/83
ENGINEER: Jeffrey T. Deutsch	PAGE: 1 of 1



DEFINE

X_FIRST=0

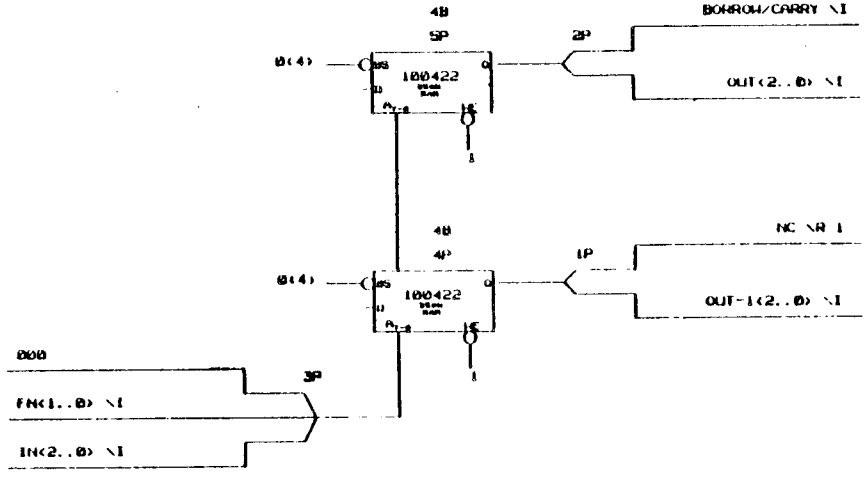
X_STEP=SIZE

DRAWING

cwp1
CWP1

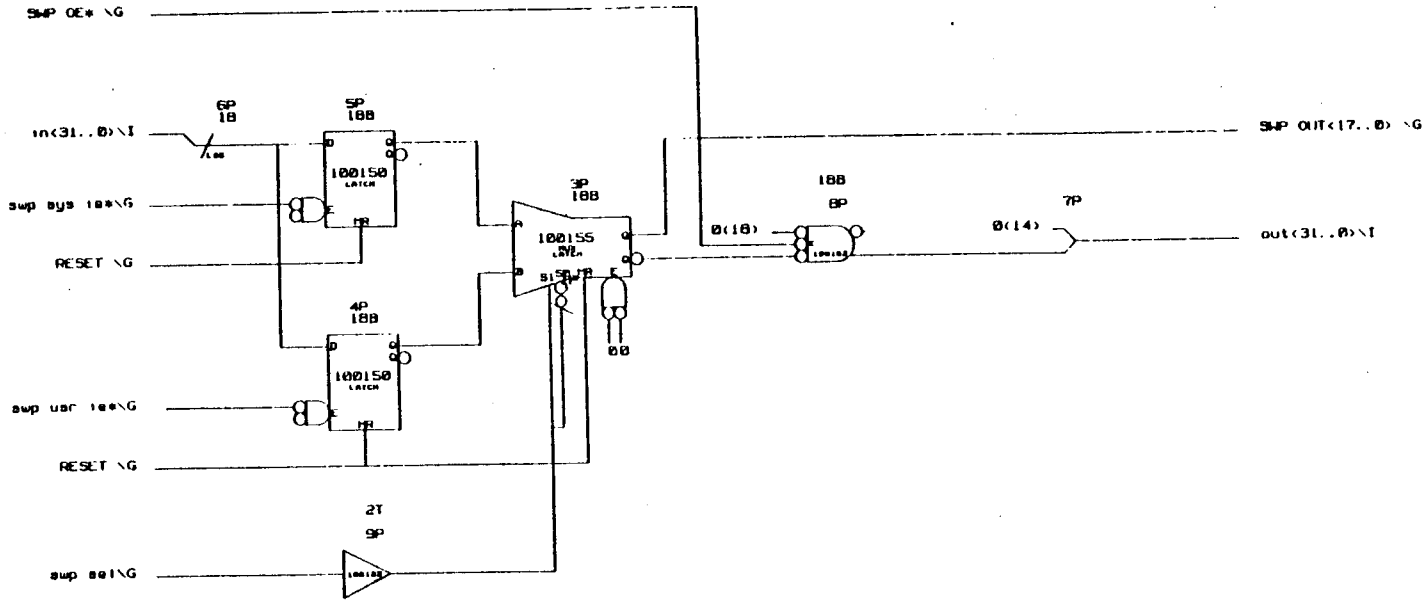
LAST_MODIFIED=Sun May 8 00:41:31 1983

TITLE: current window pointer low bits	DATE: 3/14/83
ENGINEER: Jeffrey T. Deutsch	PAGE: 1 of 1



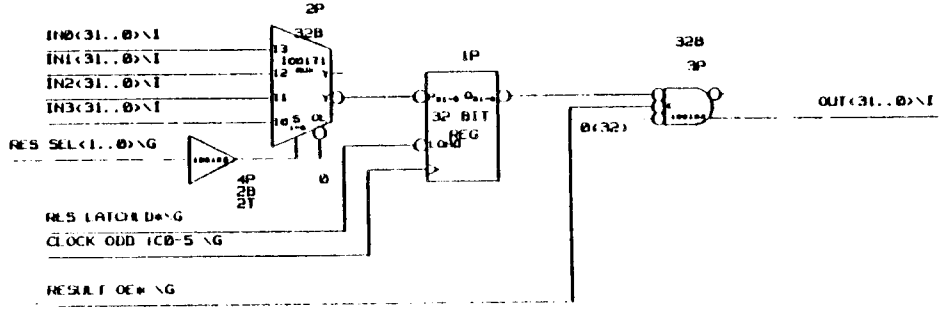
DEFINE	DRAWING
X_FIRST=Ø	ADDSUB
X_STEP=SIZE	addsub
	LAST_MODIFIED=Sun May 8 00:42:31 1983

TITLE: addsub	DATE: 3/15/83
ENGINEER: Jeffrey T. Deutsch/Rich Blomseth	PAGE: 1 of 1



DEFINE	DRAWING
X_FIRST=0	SAVED WINDOW POINTER
X_STEP=SIZE	SWP
	LAST_MODIFIED=Mon May 9 15:18:18 1983

TITLE: Saved Window Pointer	DATE: 3/13/83
ENGINEER: Jeffrey T. Deutsch/RAB	PAGE: 1 of 1



DEFINE

X_FIRST=0

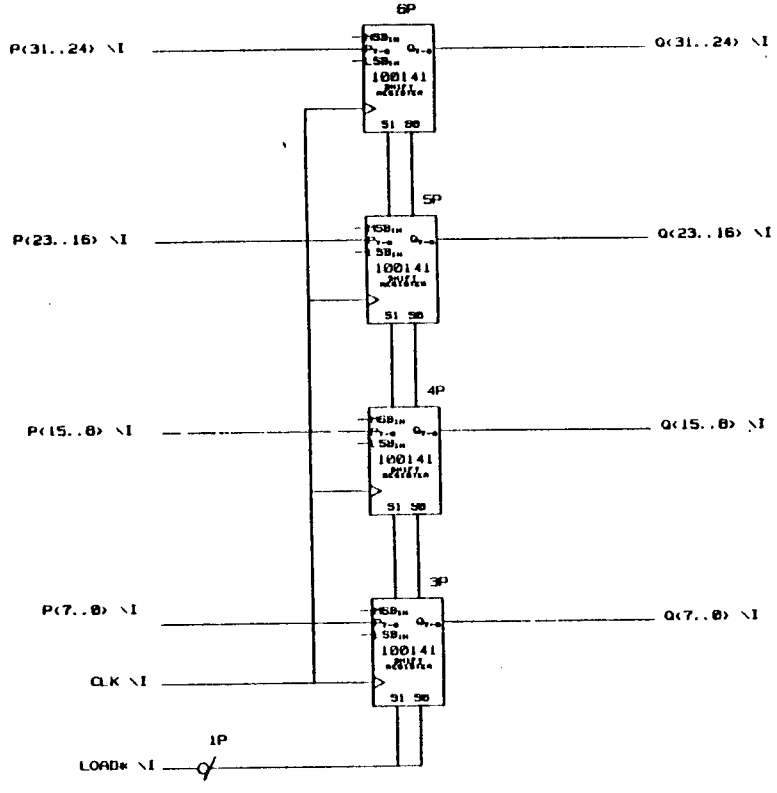
X_STEP=SIZE

DRAWING

RESULT LATCH
RLATCH

LAST_MODIFIED=Mon May 9 15:34:23 1983

TITLE: Result Latch	DATE: 3/17/83
ENGINEER: Jeffrey T. Deutsch	PAGE: 1 of 1



DEFINE

X_FIRST=0

X_STEP=SIZE

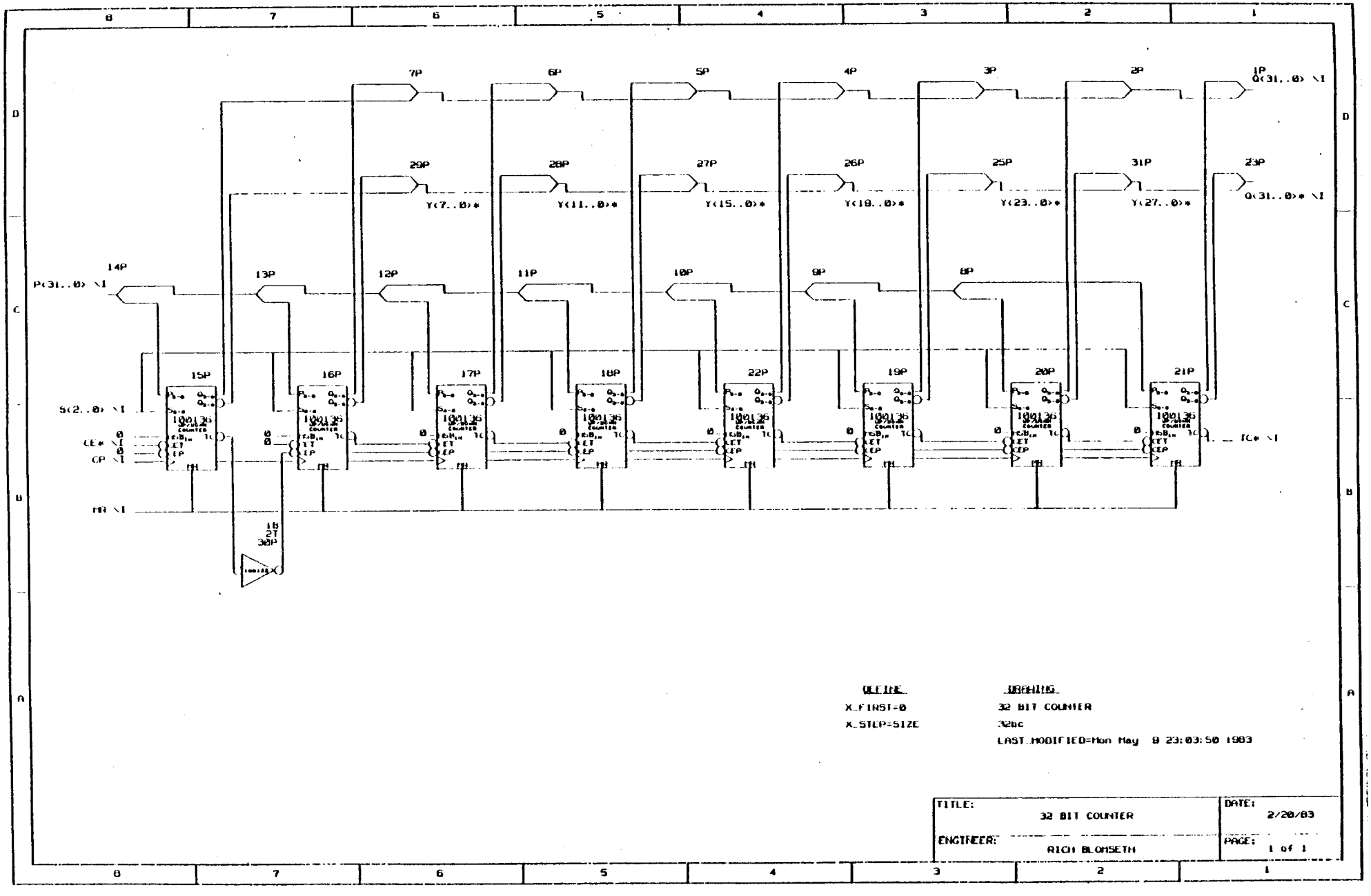
DRAWING

32 BIT REGISTER

32BR

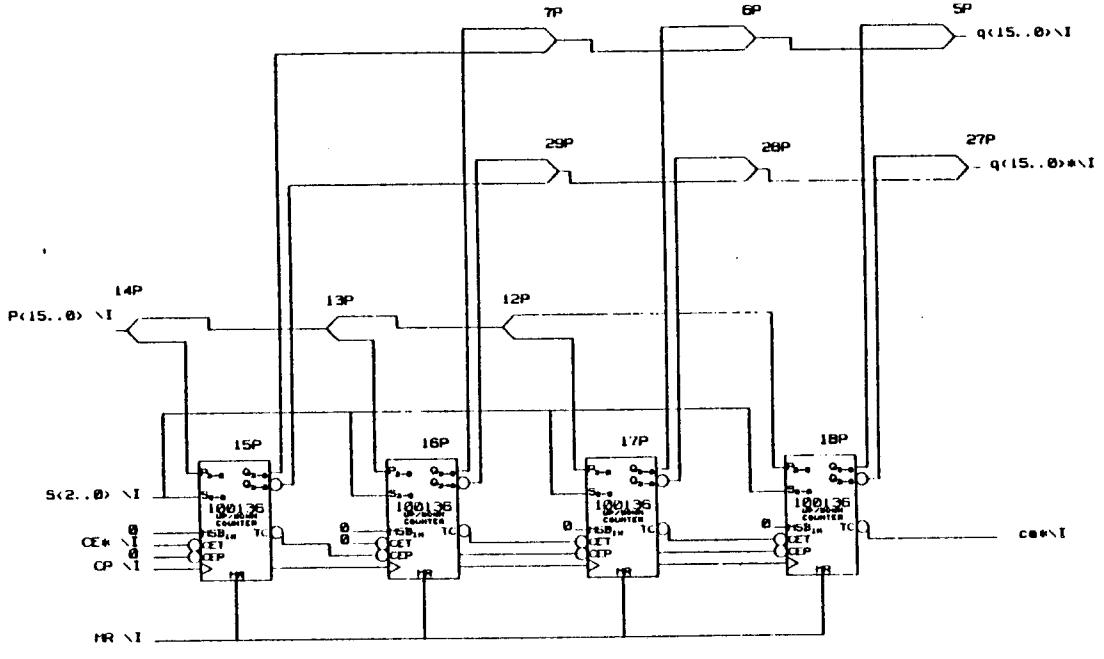
LAST_MODIFIED=Sun May 8 00:48:33 1983

TITLE: 32 BIT REGISTER	DATE: 3/11/83
ENGINEER: RICH BLOMSETH AND JEFF DEUTSCH	PAGE: 1 of 1



DEFINE	DRAWING
X_FIRST=0	32 BIT COUNTER
X_STEP=SIZE	32bc
	LAST_MODIFIED=Mon May 9 23:03:50 1983

TITLE: 32 BIT COUNTER	DATE: 2/20/83
ENGINEER: RICH BLOMSETH	PAGE: 1 of 1



DEFINE

X_FIRST=0

X_STEP=SIZE

DRAWING

18 BIT COUNTER

18bc

LAST_MODIFIED=Sun May 8 00:50:22 1983

TITLE: 18 BIT COUNTER	DATE: 3/12/83
ENGINEER: RICH BLOMSETH and Jeff Deutsch	PAGE: 1 of 1

APPENDIX F. DAPL MICROCODE LISTING

```
**
** RISC/E II Phase 3 Decode RAM microcode.
**
** 4/18/83
**

DESCRIPTION
MICROMEMORY_IS 256 WORDS_BY 52 BITS;
INTERLIST HEX;
NUMBER_BITS HIGHTOLOW;

DEFINE true AS 1.;
DEFINE false AS 0.;

FIELDS   pclbToRbusOE  WIDTH 1  DEFAULT 0.,
     pccbToRbusOE  WIDTH 1  DEFAULT 0.,
     imlShortImmed  WIDTH 1  DEFAULT 0.,
     imlLongImmed  WIDTH 1  DEFAULT 0.,
     ragenRset  WIDTH 1  DEFAULT 0.,
     sregSel          WIDTH 1  DEFAULT 0.,
     calliOpcode      WIDTH 1  DEFAULT 0.,
     forwardEnable    WIDTH 1  DEFAULT 1.,
     pipeControl2     WIDTH 1  DEFAULT 0.
     WITH (phase6Write = 0.,
          noPhase6Write = 1.),
     pipeControl10    WIDTH 2  DEFAULT 0.
     WITH (normal = 0.,
          suspend1 = 2.,  (* aligned load & store *)
          suspend2 = 3.), (* unaligned load & store *)
     spare2           WIDTH 2  DEFAULT 0.,
     shIsel       WIDTH 2  DEFAULT 0.
     WITH (shIRbus = 0.,
          shIRf0 = 1.,
          shIRf1 = 2.,
          shIIm = 3.),
     shASel           WIDTH 2  DEFAULT 1.
     WITH (shARbus = 0.,
          shARf1 = 1.,
          shAIm = 2.),
     spare3           WIDTH 1  DEFAULT 0.,
     sregIE           WIDTH 1  DEFAULT 0.,
     aluAIE           WIDTH 1  DEFAULT 1.,
     aluBIE           WIDTH 1  DEFAULT 1.,
     shILE            WIDTH 1  DEFAULT 1.,
     shAE         WIDTH 1  DEFAULT 1.,
     shIE         WIDTH 1  DEFAULT 1.,
     shSigned     WIDTH 1  DEFAULT 0.,
     spare4           WIDTH 3  DEFAULT 0.,

**
** The following signals are latched by the Phase 4 latch for
** use during phase 4.
**
```

```
    shS         WIDTH 6 DEFAULT 0.,   (* ** ? ** *)
    shRight             WIDTH 1 DEFAULT 0.,
    aluFn               WIDTH 4 DEFAULT 0.
    WITH (aluBcdAdd = 0.,
          aluBcdSub = 1.,
          aludBcdSubI = 2.,
          aluBcdNegB = 3.,
          aluAdd = 4.,
          aluSub = 5.,
          aluSubI = 6.,
          aluNegB = 7.,
          aluXnor = 8.,
          aluXor = 9.,
          aluOr = 10.,
          aluA = 11.,
          aluNotB = 12.,
          aluB = 13.,
          aluAnd = 14.,
          aluLow = 15.),
    sysCwphFn       WIDTH 3 DEFAULT 7.
    WITH (sysCwphLoad = 0.,
          sysCwphDecr = 4.,
          sysCwphClear = 5.,
          sysCwphIncr = 6.,
          sysCwphHold = 7.),
    usrCwphFn       WIDTH 3 DEFAULT 7.
    WITH (usrCwphLoad = 0.,
          usrCwphDecr = 4.,
          usrCwphClear = 5.,
          usrCwphIncr = 6.,
          usrCwphHold = 7.),
    cwplFn              WIDTH 2 DEFAULT 2.
    WITH (cwplIncr = 1.,
          cwplHold = 2.,
          cwplDecr = 3.),
    resSel              WIDTH 2 DEFAULT 0.
    WITH (resultAlu = 0.,
          resultCbus = 1.,
          resultRfl = 2.,
          resultShift = 3.),
    resLatchLdWIDTH 1 DEFAULT 1.,
    aluCarryIn WIDTH 1 DEFAULT 0.,
    spare5              WIDTH 1 DEFAULT 0.;

MICROPROGRAM
(* NOP *)
     noPhase6Write;
     noPhase6Write;
     noPhase6Write;
     noPhase6Write;
(* ADD *)
     aluAdd, resultAlu;
     aluAdd, resultAlu;
     aluAdd, resultAlu;
```

```
      aluAdd, resultAlu;
(* ADDC *)
      aluAdd, resultAlu;
      aluAdd, resultAlu;
      aluAdd, resultAlu;
      aluAdd, resultAlu;
(* SUB *)
      aluSub, resultAlu;
      aluSubI, resultAlu;
      aluSub, resultAlu;
      aluSubI, resultAlu;
(* SUBC *)
      aluSub, resultAlu;
      aluSubI, resultAlu;
      aluSub, resultAlu;
      aluSubI, resultAlu;
(* AND *)
      aluAnd, resultAlu;
      aluAnd, resultAlu;
      aluAnd, resultAlu;
      aluAnd, resultAlu;
(* OR *)
      aluOr, resultAlu;
      aluOr, resultAlu;
      aluOr, resultAlu;
      aluOr, resultAlu;
(* XOR *)
      aluXor, resultAlu;
      aluXor, resultAlu;
      aluXor, resultAlu;
      aluXor, resultAlu;
(* SRL *)
      shIRf0, shARf1, shRight = true, shSigned = false, resultShift;
      shIRf0, shARf1, shRight = true, shSigned = false, resultShift;
      shIRf0, shARf1, shRight = true, shSigned = false, resultShift;
      shIRf0, shARf1, shRight = true, shSigned = false, resultShift;
(* SRA *)
      shIRf0, shARf1, shRight = true, shSigned = true, resultShift;
      shIRf0, shARf1, shRight = true, shSigned = true, resultShift;
      shIRf0, shARf1, shRight = true, shSigned = true, resultShift;
      shIRf0, shARf1, shRight = true, shSigned = true, resultShift;
(* SLX *)
      shIRf0, shARf1, shRight = false, shSigned = false, resultShift;
      shIRf0, shARf1, shRight = false, shSigned = false, resultShift;
      shIRf0, shARf1, shRight = false, shSigned = false, resultShift;
      shIRf0, shARf1, shRight = false, shSigned = false, resultShift;
```

```
**
** RISC/E II Phase 4 Decode RAM microcode.
**
** 4/17/83
**

DESCRIPTION
MICROMEMORY_IS 128 WORDS_BY 30 BITS;
INTERLIST HEX;
NUMBER_BITS HIGHTOLOW;

DEFINE true AS 1.;
DEFINE false AS 0.;

FIELDS   spare1          WIDTH 3  DEFAULT 0.,
     camSel          WIDTH 1  DEFAULT 0.
     WITH (camPC = 0.,
          camRbus = 1.),
     pswIsel         WIDTH 1  DEFAULT 0.
     WITH (pswSys = 0.,
          pswIn = 1.),
     cwpISel         WIDTH 2  DEFAULT 0.
     WITH (cwpIIn = 0.,
          cwpISys = 2.,
          cwpIUsr = 3.),
     swpSel          WIDTH 1  DEFAULT 0.
     WITH (swpSys = 0.,
          swpUsr = 1.),
     caILE           WIDTH 1  DEFAULT 0.,
     loadPC          WIDTH 1  DEFAULT 0.,
     pccbToCbusOE  WIDTH 1  DEFAULT 0.,
     regAsMemEnable      WIDTH 1  DEFAULT 0.,
     weCache         WIDTH 1  DEFAULT 0.,
     cdIIE       WIDTH 1  DEFAULT 0.,
     csRegfile   WIDTH 1  DEFAULT 1.,
     shOLE           WIDTH 1  DEFAULT 0.,
     aluOutLatch     WIDTH 1  DEFAULT 1.,
     cwpIOE          WIDTH 1  DEFAULT 0.,
     swpOE           WIDTH 1  DEFAULT 0.,
     spare2          WIDTH 1  DEFAULT 0.,
     resultOE    WIDTH 1  DEFAULT 1.,
     pswOE           WIDTH 1  DEFAULT 0.,
     pswSysFlagsLoad      WIDTH 1  DEFAULT 0.,
     pswAluFlagsLoad      WIDTH 1  DEFAULT 0.,
     sysCwpIIE WIDTH 1  DEFAULT 0.,
     usrCwpIIE WIDTH 1  DEFAULT 0.,
     sysCwphOE       WIDTH 1  DEFAULT 0.,
     usrCwphOE       WIDTH 1  DEFAULT 0.,
     swpSysIE   WIDTH 1  DEFAULT 0.,
     swpUsrIE   WIDTH 1  DEFAULT 0.;

MICROPROGRAM
(* NOP *)
     ;
```

```
;
(* ADD *)
    regAsMemEnable = true, aluOutLatch = true;
    regAsMemEnable = true, aluOutLatch = true;
(* ADDC *)
    regAsMemEnable = true, aluOutLatch = true;
    regAsMemEnable = true, aluOutLatch = true;
(* SUB *)
    regAsMemEnable = true, aluOutLatch = true;
    regAsMemEnable = true, aluOutLatch = true;
(* SUBC *)
    regAsMemEnable = true, aluOutLatch = true;
    regAsMemEnable = true, aluOutLatch = true;
(* AND *)
    regAsMemEnable = true, aluOutLatch = true;
    regAsMemEnable = true, aluOutLatch = true;
(* OR *)
    regAsMemEnable = true, aluOutLatch = true;
    regAsMemEnable = true, aluOutLatch = true;
(* XOR *)
    regAsMemEnable = true, aluOutLatch = true;
    regAsMemEnable = true, aluOutLatch = true;
(* SRL *)
    regAsMemEnable = true, shOLE = true;
    regAsMemEnable = true, shOLE = true;
(* SRA *)
    regAsMemEnable = true, shOLE = true;
    regAsMemEnable = true, shOLE = true;
(* SLX *)
    regAsMemEnable = true, shOLE = true;
    regAsMemEnable = true, shOLE = true;
(* LDHI *)
    regAsMemEnable = false, shOLE = true;
    regAsMemEnable = false, shOLE = true;
(* CALLX *)
    regAsMemEnable = true, aluOutLatch = true, shOLE = true;
    regAsMemEnable = true, aluOutLatch = true, shOLE = true;
```