An Arithmetic-Stack Processor for High Level Language Execution

Rodney M. Goodman

Department of Electrical Engineering (116-81)
California Institute of Technology
Pasadena, CA, USA 91125
Tel. (818)-356-3677
E-mail: rogo@csvax.caltech.edu

Anthony J. McAuley[†]

Bell Communications Research (MRE-2M258)
445 South Street
Morristown, NJ, USA 07960-1910
Tel. (201)-829-4698
E-mail: mcauley@faline.bellcore.com

Abstract

We describe a 32-bit Arithmetic-Stack Processor, the ASP chip, designed to be a simple, flexible, yet powerful building block for a high level language computer.

The ASP is a 3 micron CMOS chip which runs high level programs at 25Mips, employing Forth as its machine code. It is less sophisticated than a RISC or CISC microprocessor, but when combined with the right hardware and compiler, will run procedural language programs more efficiently. The ASP is similar to a bit-slice or building block CPU, though it has a very different architecture. It therefore requires extra hardware to build a complete system, and so it would be mainly used where speed is critical. However, compared with current bit-slice or building block computers, the hardware and software design are greatly simplified.

1. Introduction

The system designer, trying to solve a particular problem, would use knowledge of algorithm complexities to find the 'best' algorithm and then translate this into a High Level Language (HLL). When performance is critical however, a standard general purpose computer may be too inefficient. In this case the designer can either try to write part of the code in machine language or try to find a computer that can run the HLL program faster. It is with the latter problem that this paper is concerned.

A computer engineer trying to help this system designer is faced with a myriad of choices. The easier and cheaper solution is to use one of the standard microprocessors, such as the Motorola 68020. But if this does not meet the speed requirements, the engineer must accept the harder and more expensive task of designing from more basic elements: such as the Advanced Micro Devices 29300 building block elements, standard chips, or design an applications specific computer chip.

[†] This work was funded by the Caltech Microsystems Group, and is covered by a Caltech patent. Dr. McAuley was involved in this work while employed at Caltech.

Our primary motivation was to build the basic hardware that would simplify the design of a wide variety of language directed architectures. The designer would follow an applications driven approach starting with a specific language and then writing the program. Only then would the designer need to be concerned with profiling the program to optimize the compiler and decide whether any hardware modification is desirable.

It is important to note from the start that the chip we designed is far below the sophistication and complexity of 32-bit microprocessors. It is just a naked stack processor. Our primary goal was to build a piece of hardware that would allow system designers to build a variety of HLL computers, customized to their needs, quickly and efficiently.

There is a well documented [1] conceptual gap between the users of computers, who typically think in terms of a High Level Language, and the computer hardware which is programmed in machine code. The design of a HLL computer to bridge this 'semantic gap' is certainly appealing. It would allow programmers to be more effective, since it is easier to see how the program actually runs in hardware. A HLL computer would also facilitate easier software debugging, greater reliability and an opportunity to customize the hardware to the problem [1].

In this paper we restrict ourselves to describing the flexibility and power of the ASP. The following sections introduce the ASP architecture, and a detailed specification sheet on the ASP chip is included as an Appendix for the use of potential users.

It is clear there must be some trade-off between simplicity, flexibility and power when designing a computer system. In section 2 we very briefly review three broad approaches to this problem. In section 3 we describe the virtual stack machine, which is the basis of our approach. Sections 4, 5 and 6 outline the ASP chip hardware, defering much of the detail to the Appendix for simplicity. Section 6 introduces the microcode, again the detail is given in the Appendix. In section 8 we outline how to use the ASP, with the aid of additional hardware and software, to build a machine dedicated to Forth. Section 8 introduces the idea of using the ASP to implement other high level languages such as 'C'. Using the ASP to implement languages other than Forth will be the subject of future papers.

2. Three Classes of Computers

In this section we briefly review three classes of computer architecture: the RISC, the CISC and the BISC (our own acronym). Though the dividing line is somewhat fuzzy, this characterization enables us to differentiate our approach from existing methods. This section does not cover any new ground, so those familiar with these three classes can skip this section.

Recently, the most popular approach has been the Reduced Instruction Set Computer (RISC). This name covers a broad spectrum of architectures, but all have a small number of simple instructions tailored towards the most used primitives of a HLL, each executing in a single clock cycle [1,2].

One approach to RISC architecture is to have a large number of registers and a fast instruction pipeline. The instructions, though directed towards a HLL, are normally of fairly low level. An example of this approach is the UC Berkeley RISC II machine [1]. A second approach to RISC is to have virtually no internal registers, making context switching much faster. The Novix NC4000 [4] is an example of this approach, which has just 40 primitives.

Our second broad classification is the Complex Instruction Set Computer (CISC) [1]. CISC's try to migrate many of the HLL constructs directly into the primitive machine instruction. The microcode, equivalent to a RISC instruction, is hidden in the on-chip sequencer. This allows a reduction in the number of instructions needed to perform a particular task, at the cost of some flexibility. A good example of a CISC is the Motorola 68020 [1].

Our final classification is the Building Block style computer (BISC). These provide very flexible building blocks, such as an ALU, sequencer, and multiplier, for high performance applications. Despite the programming problems associated with low level, complex microinstructions, they remain popular because they can out-perform standard microprocessors. A good example of a BISC is the Advanced Micro Devices 29300 family [5]. We also include in this class computers built from standard TTL or custom/semi-custom chips (see also pages 5–9 of reference [3]).

Even though RISC and CISC architectures continue to improve, and indeed be combined, they are often too inefficient for high performance applications. While the throughput achievable with the BISC approach is potentially much better, it is expensive and takes even experienced designers a long time to complete. What is required is a combination of this power and flexibility, together with simplicity in both hardware and software.

3. The Virtual Stack Machine

When a HLL is compiled to run on a microprocessor, there is normally an intermediate language between the source code (e.g. 'C') and the machine language (e.g. 68020 machine code). This internal language is most efficient [7] when it is based on a Reverse Polish machine, that is, on a virtual stack machine. In this case the operators occur in the same order in which they are employed, so they are quicker and easier to run.

The basic building blocks of a 'typical' virtual stack machine are: an Arithmetic/Logic Unit (ALU), an arithmetic stack, a return stack, an instruction fetch unit (IFU), a microcode-sequencer and some data and instruction memory.

The function of the ALU and arithmetic stack are the execution of the language primitives. The function of the IFU and return stack are to get the address of the next instruction and perform the stack frame management of the subroutine calls. The exact function and interaction between the ALU and IFU blocks will depend on the language being implemented. But, we can make two important observations at this stage. First, the ALU and IFU though functionally very different, actually require similar hardware. Secondly, they can operate fairly independently, for example, the IFU could work concurrently or even asynchronously with the ALU, except when the two need to communicate.

It is now possible to put all of the above onto a single chip, but with limitations on the size of stack and a general lack of flexibility. In order to fulfill our goal of flexibility, we have mapped only the critical, 'lowest common denominator,' hardware elements onto a fast custom VLSI chip. This approach, described in section 7, gives greater freedom for applications driven solutions.

The chip is designed to act as the core hardware needed to implement both the ALU and IFU. Because their common functions are an Arithmetic Unit and a Stack Manipulator we have dubbed our device the Arithmetic Stack Processor, or ASP.

4. The Arithmetic Stack Processor

The 32 bit ASP can be logically split into a Central Processing Unit (CPU) and a Stack Pointer Unit (SPU). These represent our choice of the two key elements to running an HLL. Figure 1 shows a block diagram of the ASP, while Figure 2 describes the function of the external pins.

The ASP is a flexible, high speed 32 bit microprogrammable device. It has the potential to perform 2^{27} different operations. Of course many of these are not meaningful. However, as the microcode directly controls the hardware, and the user has direct access to this it is easy to find primitives which can be used to optimize different applications. Microcode is loaded via pins M26–M0, with the A31–A0 bus acting as the port for external data transfers. The external 'stack RAM' is read or written via D31–D0, with the address supplied on pins L23–L0. The addition of

external RAM thus transforms the ASP into a true stack processor. The operation and management of this RAM is invisible to the rest of the system.

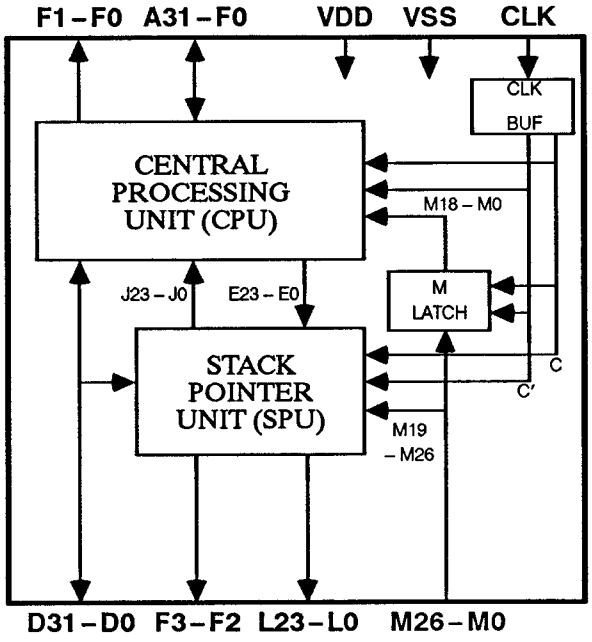


Figure 1. ASP Functional Block Diagram.

Simplicity, flexibility and power are the three keys to the ASP. The simplicity is in the architecture which consists of just a few registers, multiplexers, and adders directly driven by microcode. The flexibility is in its ability to perform potentially 2^{27} different microinstructions. The power of the ASP comes from both the high level of the microinstructions, which are higher than standard microprocessor machine code, and the speed at which it can execute them.

There is one level of pipelining on the chip, used in order to minimize the speed requirement of external RAM. The SPU calculates the address one cycle before the CPU needs to read or write the corresponding data. To achieve this, the microcode goes directly into the SPU, allowing it to add the contents from one of its stack pointers to any offset required, and latch the result onto the L-bus latch. The M-latch (see Figure 1) delays the microcode one clock cycle before it reaches the CPU, allowing it to read or write from the external RAM with the address already waiting on the L-bus.

Signal	Description
M26 - M0	Microcode input bus
A31 - A0	Bidirectional, tristate system bus (Inverted signals)
D31 - D0	Bidirectional, tristate stack RAM data bus (Inverted signals)
L23 - L0	Stack RAM address bus
F3	Stack underflow
F2	Stack overflow
F1	ALU numeric overflow
F0	ALU zero detect
CLK	System clock input
VDD(2)	Power input, 5 volts
VSS(2)	Power input, 0 Volts (GND)

Figure 2. ASP Pin Description.

Figures 3 and 4 shows a simplified block diagram of the SPU and CPU. Most blocks actually have a separate pathway to the other blocks, though for simplicity this is just labeled as a routing network.

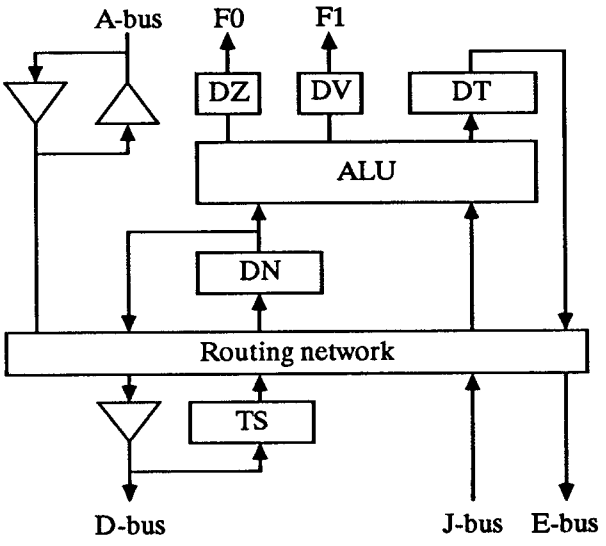


Figure 3. The Central Processing Unit (CPU).

The CPU, shown in Figure 3, consists of an ALU, the top two stack registers (DT and DN), a transparent latch (TS), two one bit flag registers (DV and DZ), tri-state buffers (the triangles), and multiplexers (shown as a routing network). The ALU is very powerful and flexible, and contains many unusual capabilities to enable most internal language primitives to run in one clock cycle (see section 6). The flags F1 and F0 tell the external sequencer whether the ALU result was zero or if an arithmetic overflow has occurred. The ALU would normally take its operands from DT, DN, but can also take them from the SPU, the D-bus (via TS) or the A-bus. Its operation is controlled by the 19 bits of the microcode in the M-latch.

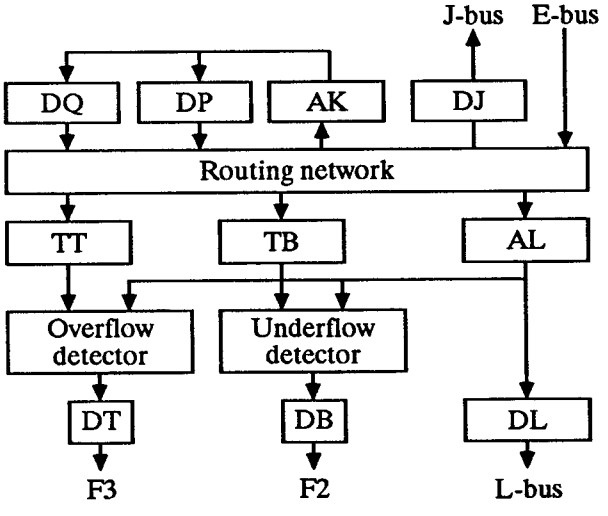


Figure 4. The Stack Pointer Unit (SPU).

The SPU, shown in Figure 4, has two stack pointer registers (DP & DQ), stack limit registers (TT & TB), buffers (DL & DJ), flag registers (DT and DB), adders (AL and AK), comparators (underflow and overflow detectors), and multiplexers (shown as a routing network). Its operation is controlled by eight microcode lines (m26–m19). The main function of the SPU is to generate the new stack address allowing the external RAM to simulate a stack. Normally (although there is in fact much flexibility) the SPU would take DP as the base address and add to it some bits of the microcode using AL. At the same time the SPU updates DP, which is normally equal to the old DP plus some microcode bits added together by AK. F3 and F2 indicate whether the address has overflowed or underflowed the limits set in TT and TB respectively.

5. The Custom CMOS Chip Design

The ASP was fabricated in 3-micron, double level metal CMOS, with less than 25,000 transistors on a 6.8-mm^2 chip, and packaged in a 124 pin grid array. By using custom design, as opposed to gate arrays, we were able to get the most out of the silicon. Yet, because of the simplicity of the architecture, it could be designed, laid out and fully tested in under a year. In this section and the next we give a brief description of the chip, the data sheet in the appendix gives more detail.

The ASP is fully static, so that it requires no minimum clock frequency or refresh, and consumes less than a Watt at 12.5 MHz. It has a single external clock line, and all data changes on the falling edge. The pins can read or write from either CMOS or TTL devices and drive up to 20pF.

Figure 5 shows a photograph of a working ASP chip. It was designed on a SUN 3/160C, running the HILO-3™ simulator and MAGIC layout editor. Testing was done with a VME hardware set-up, with the chip connected through ports to the host VME computer. The test program on the host was written in Forth, and allowed microcode to be sent to the chip, and input and output data to be manipulated. This software based testing philosophy is very fast and flexible.

For speed, the three adders and two comparators in the ASP were designed with a full Carry-Look-Ahead. The number of transistors on the ASP could be reduced to below 20,000 by replacing these designs by simpler Manchester Carry Chains **[9]**. Other simplifications could reduce the transistor count to around 10,000 transistors, without significant performance degradation, making it an ideal candidate for GaAs. However, some re-design to exploit GaAs's strengths and weaknesses would be required **[6]**, such as putting some form of on-chip cache 'stack RAM' to compensate for the relatively slow off-chip speed.

6. The Microcode

Most internal language instructions can be concocted by setting some combination of the 27 microcode lines to 0 and 1. For example in one clock cycle the ASP can:

a) Duplicate any of the top three elements from the stack.

b) Drop any of the top three elements from the stack.

c) Compare two numbers, leaving an all-ones or zero flag on top of the stack to say if they were larger, smaller, or equal.

d) Add, Subtract or do any logical operation on the top two elements of the stack.

In two clock cycles (two separate microinstructions) the ASP can:

a) Do a double precision add, with 64 bit operands.

b) Copy an item from anywhere on the stack to the top of stack.

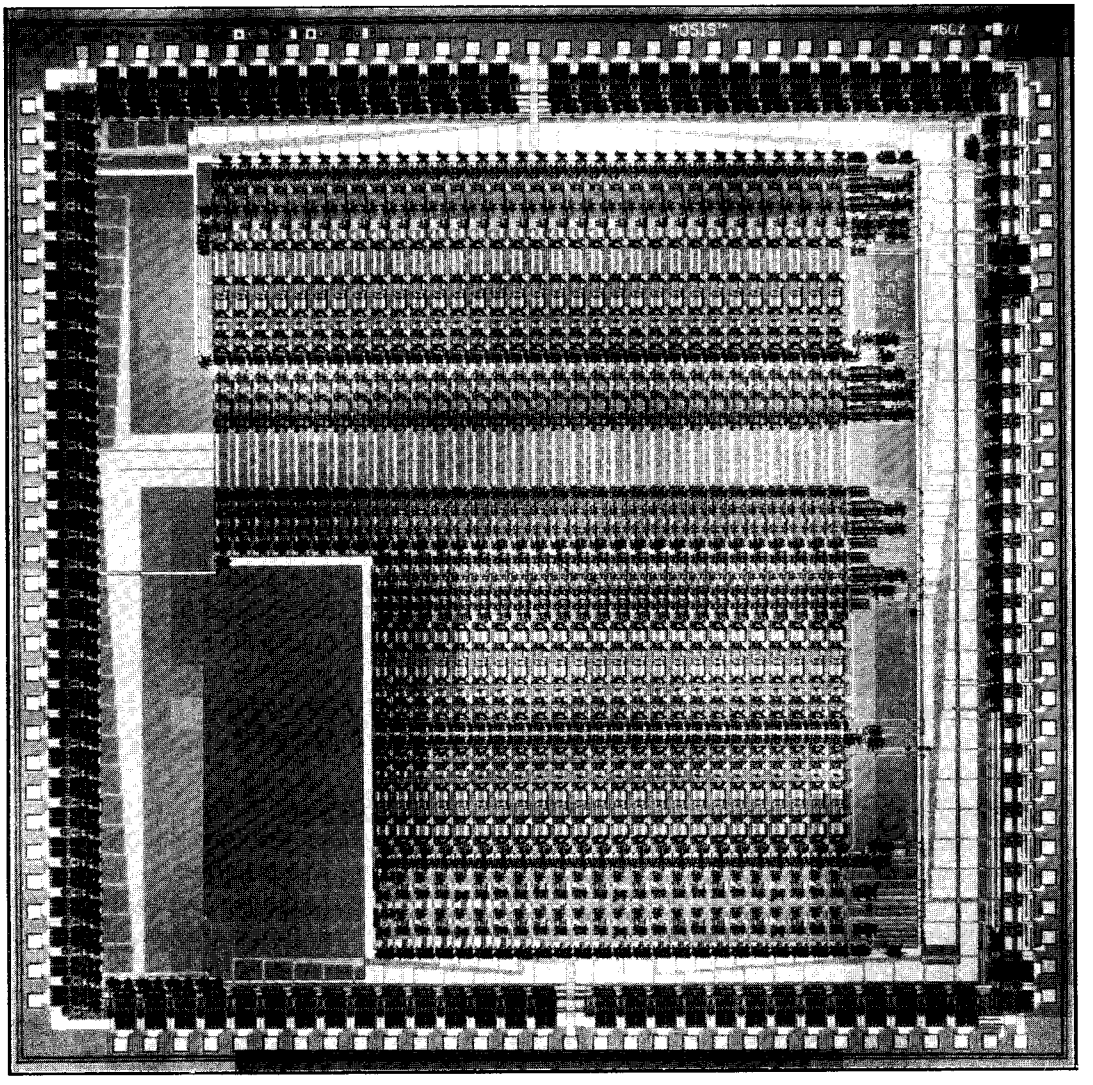


Figure 5. Photograph of the ASP Chip in 3 micron CMOS.

The microcode does not stick rigidly to any internal language, though it was optimized for procedural languages such as Forth and C. The microcode suitable for a Forth set of primitives is covered in detail in the Appendix.

All the 2^{27} possible instructions execute in a single clock cycle, giving a simulated performance of 25Mips in 3 micron CMOS, with a 25MHz clock. Because of testing limitations, the ASP has so far only been run at 12.5 MHz. But note, these are stack based HLL instructions, not just machine code operations.

7. A Forth Machine

This section will briefly describe a Forth machine, built around two ASP chips. A simple such architecture is shown in Figure 6. This is in fact very similar to the virtual machine described in section 3, the ASP-X and RAM-X together do the arithmetic execution, while ASP-T and RAM-T perform the stack frame manipulation.

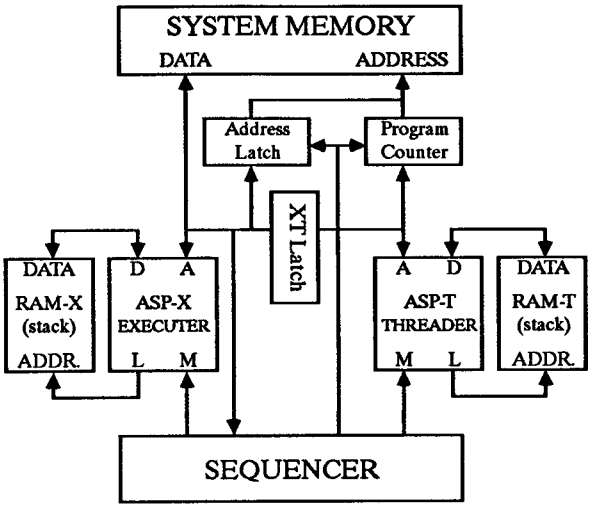


Figure 6. A Simple Virtual Stack Machine.

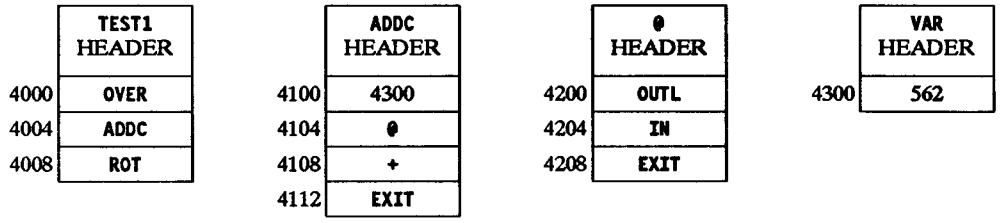
Forth represents a suitable internal language for compiling other procedural languages, as well as being a HLL in its own right. Forth is built from 'words,' each defined in terms of other words, and so on, until a primitive word definition is encountered. One of the reasons for Forth's popularity is that it can be completely defined with less than 30 primitive instructions. In our case these Forth primitives are just defined in terms of the microcode required for the two ASP chips. For performance reasons we found 100 primitives a good number, though the choice depends on the application and can easily be changed.

When Forth executes, it does so by threading through its defined 'words.' The instruction pointer contains the address of the next word. If it is a primitive, the code simply tells the sequencer which microcode to feed to the two ASP chips. When a non-primitive is encountered the code tells the sequencer to load a new address into the instruction pointer, saving the old one on the return stack; or to load the instruction pointer with the top of the return stack Thus each 'word' steps through its instructions by successively pointing to the next word in memory, until a non-primitive instruction is encountered. The current address is then saved on the return stack and the new word can begin stepping through its own instructions. This goes on until one 'word' finishes and exits back up to its calling 'word.' This hierarchical threading, as this calling and exiting is termed, is the responsibility of ASP-T (see Figure 6). The saved addresses are stored in RAM-T and the in-line stepping is done by the program pointer.

The process is best illustrated by a small example. At the top of Figure 7 the memory used for defining four words (`TEST1`, `ADDC`, `@` and a variable) are represented. Below this are six diagrams used to represent the state of the executor and threader as we thread through the memory, in order to execute the word `TEST1`. Using Figure 6 as a reference, AL and IP represent the contents of the Address Latch and Program Counter, 'p1–p2' and 'r1–r2' represent the two stack elements stored on chip in the ASP-X and ASP-T respectively, while p3–p4 and r3–r4 represent the next two RAM locations from the current L-bus address in the RAM-X and RAM-T respectively.

One very important fact that emerges from this example is that threading takes at most one clock cycle. The ASP can therefore execute direct threaded code with an efficiency approaching that of in-line code. `EXIT`s can take no time just by adding a bit field to the primitives to tell the ASP-T to do an `EXIT`, while ASP-X concurrently executes the primitive. In fact even threading can normally be done in 'no time' if the sequencer can pipeline its primitives.

The hardware shown in Figure 6 represents a minimal high performance system. However, the flexibility of the ASP provides for easy extensions. For example, the basic two stack pointer system in the ASP allows the executer to maintain say a floating point stack separate from the integer stack. The threader can also perform complex pointer manipulations and arithmetic with the two stack pointers. Addition of extra hardware such as a fast multiplier or floating point device is easy, and turns the ASP into a very powerful element. Multiple such ASP chips can be used in an architecture to achieve very high performance. On the software side it is possible to use multiple ASPs to run optimized subsets of the language concurrently. The ASP thus allows the compiler and/or external hardware to be tailored towards any particular language – thus producing a virtual machine for the language.



System Memory (Containing Forth Words)

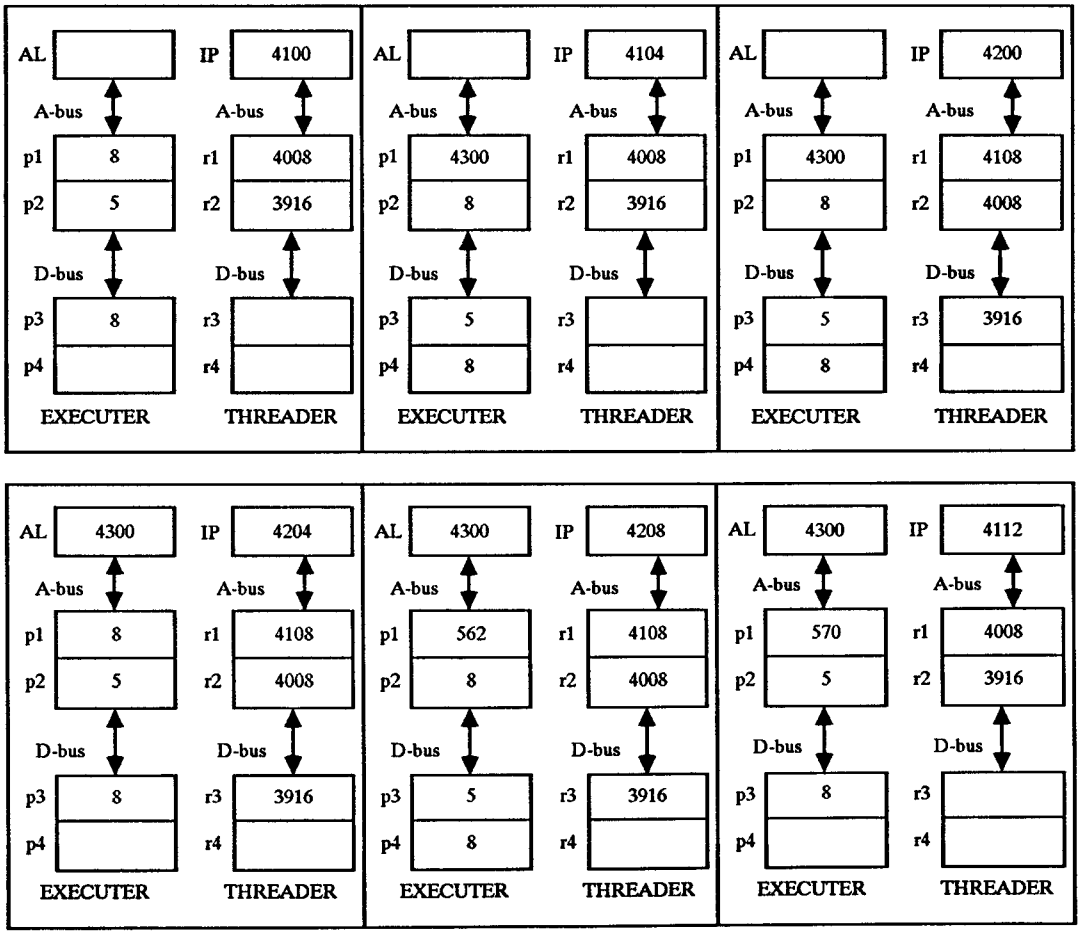


Figure 7. Effects on P & R Stacks while Executing a Forth Program.

8. Other Languages

In this section we give a flavor of how languages other than Forth could be implemented with the ASP. These ideas will be fully developed in subsequent papers.

Consider how the ASP architecture shown in Figure 6 can be used to run the C language [8]. Given the following C statement: IF (a == 0) b = c + d . The equivalent internal language primitives for this in Reverse Polish might be:

```
a FETCH 0= IF c FETCH d FETCH + b STORE THEN
```

The two statements perform the same task. A compiler would have little trouble going from the C to the internal language. Then, as shown in the last section we can very efficiently run the resulting internal language program. Clearly there is room for improvement, even in this simple illustration. For example, the program would be quicker if the variables were stored on the stack instead of FETCHed and STOREd.

It must be made clear, however, that the C-machine is not just a microprocessor with a C-compiler stuck on. In order to maximize efficiency we must employ a different set of primitives. The choice of primitives will depend on which parts of the language are required to run most efficiently, but the ASP is flexible enough to already have those primitives.

It is possible to write an optimized compiler for any language (for example Lisp, SmallTalk or a DSP language). However, as we move away from a procedural language, it is no longer sufficient to change the primitives. The ASP architecture itself will begin to limit the performance. The reasons for this are complex and more work is needed in this area to see what hardware changes are necessary.

Our approach to this problem will be the same as for the ASP chip design: applications driven. Mapping down from the language to hardware using a profile of typical program primitives. Take for example a DSP program. The compiler could be changed to allow the second stack pointer to step through coefficients, without changing the main stack pointer. Additional architecture specifically for a DSP-machine might include hardware for doing very fast convolution.

9. Conclusions

The ASP has some elements of standard microprocessors, but without their inflexible architecture or relatively slow HLL execution. It is similar to BISC computers, without their complexity. It's design philosophy closely resembles the RISC machines, yet it has the potential to execute 2^{27} different microintructions, and has few internal registers. The latter makes context switches (for multiuser and multitasking application) much easier.

The basic ASP microcode instructions are at a higher level than any microprocessor machine code. In fact they blur the hardware and software interface, allowing better use of computer system resources. Each microcode instruction runs in one clock cycle, which is 25MHz in 3 micron CMOS.

The ASP is a very flexible building block which can be tailored to the implementation of special purpose or standard high level languages such as Forth and C. In this paper we have outlined the basic ASP chip architecture. In subsequent papers we will explore in detail the design and operation of Forth, C, and special purpose computers built out of ASP chips. We encourage experimenters to use ASP chips in their designs, and will provide sample chips for their use.

10. References

[1] A. Silbey, V. Milutinovic & V. Mendoza-Grado. "A Survey of Advanced Microprocessors and HLL Computer Architectures." *Computer*, August 1986, pp. 72–85.

[2] R. Weiss. "RISC Processors: The New Wave in Computer Systems," *Computer Design*, May 15, 1987, pp. 53–73.

[3] M. Starling. *The Journal of Forth Applications and Research – Special on Forth Machines*, Vol.2, No.1, 1984.

[4] J. Golden, C. Moore & L. Brodie. "Fast Processor Chip Takes its Instructions Directly from Forth." *Electronic Design*, March 21, 1985, pp. 127–135.

[5] P. Chu & B.J. New. "Microprogrammable Chips Blend Top Performance with 32-bit Structure." *Electronic Design*, November 15, 1984.

[6] V. Milutinovic, D. Fura, W. Helbig & J. Linn. "Architecture/Compiler Synergism in GaAs Computer Systems." *IEEE Computer*, May 1987, pp. 72–93.

[7] P.J. Brown. *Writing Interactive Compilers and Interpreters*, Wiley, 1979.

[8] A. Kelley & I. Pohl. *A Book on C*, Benjamin/Cummings, Menlo Park, Ca., 1984.

[9] K. Hwang. *Computer Arithmetic: Principles, Architecture and Design*, Wiley, New York 1979.

Dr. Goodman received a B.Sc. in EE from Leeds University in Yorkshire, UK in 1968 and a Ph.D. in EE from the University of Kent in Canterbury in 1975. He joined the faculty of the Department of Electrical Engineering at the California Institute of Technology as an associate professor in 1985. His research has spanned cryptography, neural nets and expert systems. He founded two companies in the UK: Electronic Automation Ltd. which specializes in robotic vision and Advanced Processor Design Ltd. who are developing high speed Forth architectures. Dr. Goodman's current research is directed at neural network VLSI architectures and autonomous real time expert systems.

Dr. McCauley received his B.Sc and Ph.D. degrees in Computer Engineering in 1981 and 1985 from the University of Hull, England. From 1985 until 1987 he was a research fellow at the California Institute of Technology working on computer architecture, coding and logic design. Since 1987 he has been with Bellcore researching high speed switching and high performance protocols. His current interests include fault tolerance, Forth machines and VLSI architecture.

ASP - ALU Stack Processor

Appendix A. Preliminary Data Sheet Ver 1.0

Distinctive Features

- 32 bit ALU
- 32 bit multiplexed system address and data bus.
- 27 independent, non-coded microcode lines.
- 32 bit stack RAM data bus.
- 24 bit stack address bus.
- 124 pin grid array package.
- 80 ns instruction cycle (with a 12.5 MHz clock).
- Low power, fully static CMOS technology.
- Microcode is a high level language, eliminating the need for assembly language.

1. General Description

The ASP is a flexible, high speed 32 bit microprogrammable device. It can perform 2^{27} different operations affecting it's stack contents, such as: manipulation, comparison, logic, arithmetic and external memory operations. Microcode is loaded via pins M26–M0, with system access via A31–A0. The 'stack RAM' is read or written via D31–D0, with the address supplied on pins L23–L0. All data changes on the falling CLK edge.

Figure 1 shows a simplified functional block diagram of the ASP, with its two major sub-blocks. The CPU contains an ALU the top two stack registers. The SPU has two stack pointers and two adders; allowing the external 'stack RAM' to simulate two stacks.

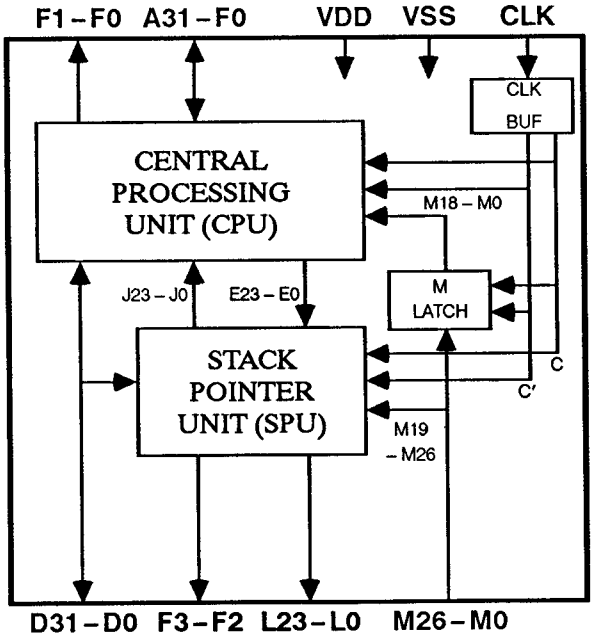


Figure 1. ASP Functional Block Diagram.

The M latch delays the external microcode (M) by one clock cycle, before it reaches the CPU. This pipeline allows the stack address to be calculated and waiting, when the CPU operation begins.

Signal	Description
M26 – M0	Microcode input bus
A31 – A0	Bidirectional, tristate system bus (Inverted signals)
D31 – D0	Bidirectional, tristate stack RAM data bus (Inverted signals)
L23 – L0	Stack RAM address bus
F3	Stack underflow
F2	Stack overflow
F1	ALU numeric overflow
F0	ALU zero detect
CLK	System clock input
VDD(2)	Power input, 5 volts
VSS(2)	Power input, 0 Volts (GND)

Table 1. Pin Description.

ASP is designed to form the main computational element(s) in a general purpose computer system. In particular, the architecture has been optimized to run stack oriented primitives. But, as the virtual machine for most procedural languages, such as C, are stack based, the ASP is a true high level language (HLL) processor. HLL programs can run on the ASP system without the normal performance degredation associated with a HLL computer implementation.

2. External Interface

The ASP is packaged in an 124 pin grid array. Table 1 descrbes the function of each pin, while appendix spec-A shows the relationship between these signal names and their pin positions (Table 2).

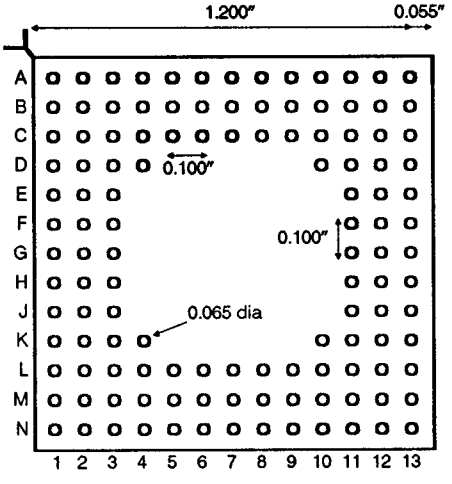


Figure 2. Pin Connections (Pins Down)

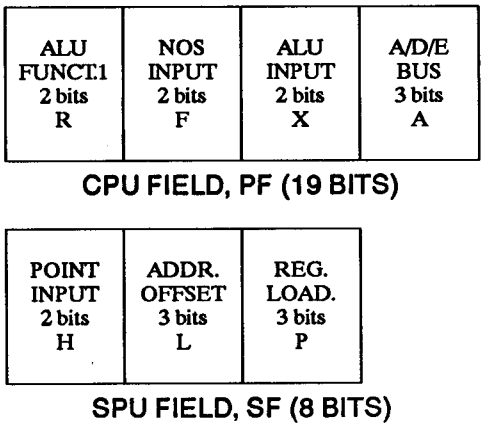


Figure 3. Instruction Format.

Select		ME-out
u15	u14	X
0	0	E
0	1	J
1	0	S
1	1	T

Select		MF-out
u13	u12	F
0	0	Y
0	1	G
1	0	S
1	1	T

Table 2. Multiplexer Truth Tables.

3. Instruction Format

The microcode (m26–m0) can be logically split into two parts: the CPU Field (CF) and the SPU Field (SF). Each field can be further subdivided into a total of 7 subfields shown in Figure 3.

4. CPU Architecture

Figure 4 shows a simplified block diagram of the Central Processor Unit (CPU). It consists of the Arithmetic and Logic Unit (ALU), a parallel multiplier (MLY) (optional), the Top Of Stack (TOS) and Next On Stack (NOS) registers (DT and DN), multiplexers and buffers.

Table 2 describes the operation of the two multiplexers MF and ME.

4.1 Multiplier.

MLY is a parallel multiplier, employing a modified Booth's algorithm. It's two 32 bit operands come from TOS and NOS.

The least significant 32 bits of the product (G) is fed into the mutiplexer MF and, if u13–u12 are set correctly, into NOS. The most significant 32 bits of the result are output in carry save format, together with a carry (CM), as K and L: i.e., as two 32 bit numbers. These two numbers (and cm) are combined in the ALU and stored in TOS.

Thus, the MLY and ALU can multiply NOS by TOS and store the 64 bit result in the same two registers.

The time taken for a multiply is 100ns, which is longer than a single clock cycle at 20MHz. Therefore, with a fast clock, the operands must remain in TOS and NOS for one clock cycle before the multiplication proper can begin.

(not implemented in version 1.0 chips)

Select Lines			Left Operand
u10	u9	u8	LO
0	0	0	Y
0	0	1	0
0	1	0	K
0	1	1	Y'
1	0	0	2
1	0	1	4
1	1	0	-2
1	1	1	-4

Y = Input.
Y' = Inverted input.
K = Multiply input.
LO = Output.
u10-u8=microcode.

Table 3. Truth Table /Left Operand

Select Lines		Right Operand
u7	u6	RO
0	0	X
0	1	0
1	0	L
1	1	X'

X = Input bus 1.
X' = Inverted X.
L = Input bus 2.
RO = Output.
u7-u6 = microcode

Table 4. Truth Table /Right Operand

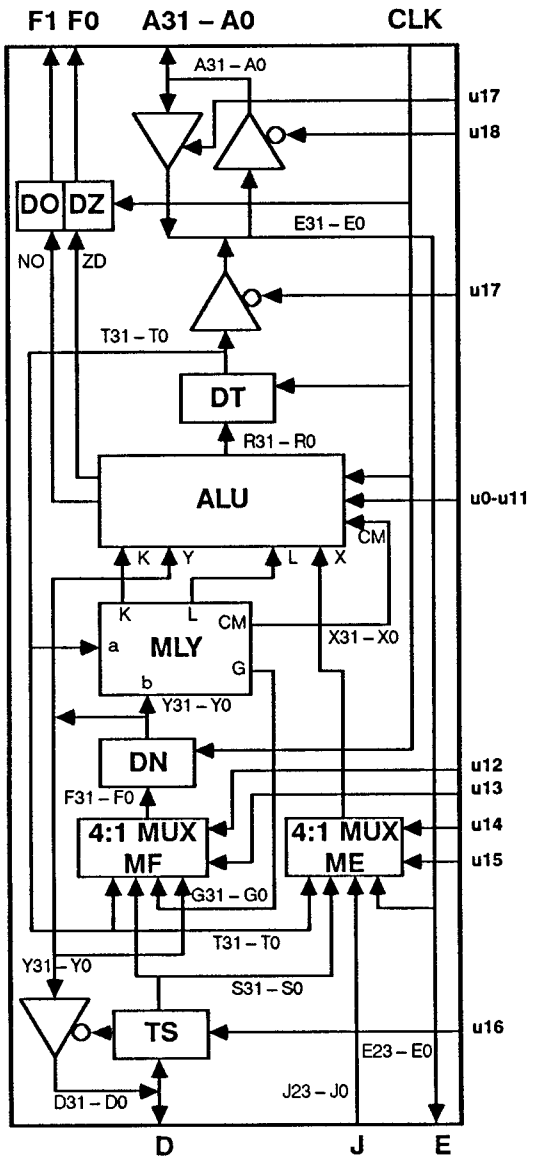


Figure 4. CPU Block Diagram.

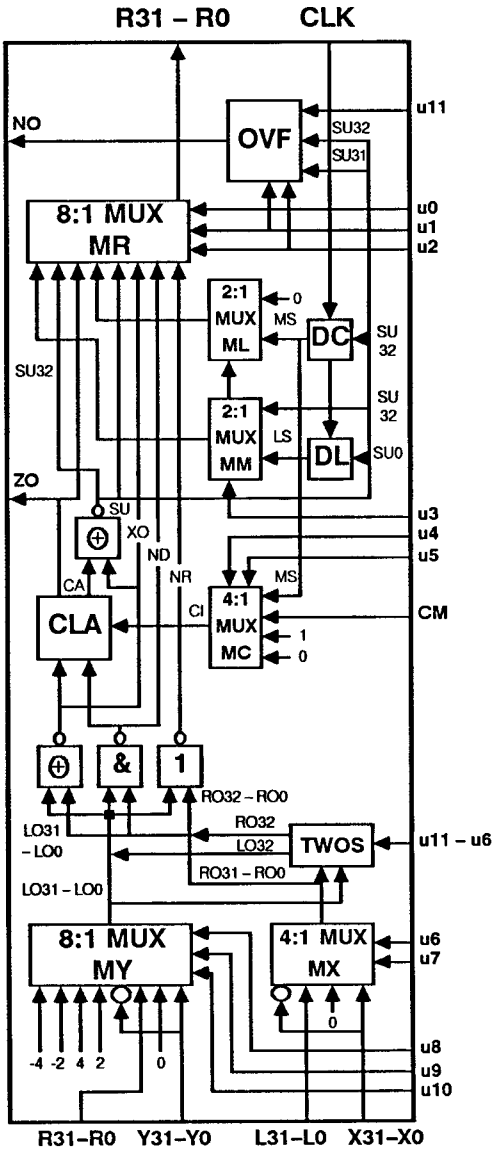


Figure 5. ALU Block Diagram.

4.2 The Arithmetic And Logic Unit.

The Arithmetic and Logic Unit (ALU) calculates a 32 bit result (R) from its X and Y (or K and L) operands, under control of twelve microcode lines (u11–u0). It performs most typical ALU operations, such as addition, subtraction, shifting and gateing. But, in addition, it can do other powerful operations: such as add a wide variety of constants, or output a truth value (all 1's or all 0's) dependent on the result of the current operation.

Select Lines			Output
u2	u1	u0	RO
0	0	0	SU
0	0	1	SL
0	1	0	SU32
0	1	1	ZO
1	0	0	NR
1	0	1	ND
1	1	0	XN
1	1	1	SR

(a)

Select Lines	Output of MUXs	
u3	ML	MM
0	0	SU32
1	MS	LS

u5	u4	CI
0	0	0
0	1	1
1	0	CM
1	1	MS

(b)

Table 5. ALU Selection Tables.

There are four subfields in the ALU microcode (Tables 3, 4, 5a and 5b), each corresponds to a major subsection of the ALU architecture shown in Figure 5. The first two subfields (selected by u10–u6) determine the left (LO) and right (RO) operands entering the rest of the ALU.

The left (LO) and right (R0) operands are the outputs of the two multiplexers MY and MX respectively. For LO, bits u10–u8 are used to select between eight possible values, as shown in Figure 3. While for RO, u7–u6 are used to select between four inputs, as shown in Table 4.

The 32 bit outputs from the multiplexers, LO and RO, are blocked up to 33 bits by the TWOS logic box. If u11=0 it will sign extend the msbs (RO32=RO31 & LO32=LO31). If u11=1, it will either force both the 33rd bits to be zero (RO32=LO32=0), or if a number is being subtracted (determined by u10–u6) set one of the 33rd bits equal to 1. Thus, u11 selects between twos complement (0) and unsigned (1) notation for all arithmetic operations.

LO and RO are combined using 32 XOR, NAND and NOR gates. The output of the first two (XO and ND) are used, together with the carry in (CI), in the Carry Look-ahead Adder (CLA). The carry (CA) out of the CLA is combined with XO to produce the arithmetic sum, SU (the sign bit, SU32 is also a function of u11-u6).

The carry into the CLA, CI, is either: MS, CM (overflow from multiplier), 0 or 1 depending on the values of u5 and u4, as shown in Table 5b. Setting CI equal to 1, allows odd valued constants to be added. For example, if u10–u8 is '111', u7–u6 is '00' and u5–u4 is '01'; the sum, SU, would be X-3 (X-4 + CI): see Tables 3, 4 and 5b.

During shift operations ML and MM provide the lsb or msb, respectively. Table 5b shows how the two values are chosen, using u3: where MS is the carry (SU32) and LS is the lsb (SU0) from the previous clock cycle.

The final output of the ALU, R31–R0, is determined by u2–u0: as shown in Table 5a. R is either a logical combination of LO31–LO0 and RO31–RO0 (XO, ND, NR), an arithmetic combination of the same (SU), a shifted version of SU (SR or SL, with the extra bit from MM or ML), or all R bits are forced equal to ZO or SU32. The latter indicates SU is a negative number (for unsigned arithmetic, this is only true for a subtraction).

Select Lines		Output
u20	u19	H
0	0	D
0	1	E
1	0	Q
1	1	P

Table 6. Truth Table for MH of SPU.

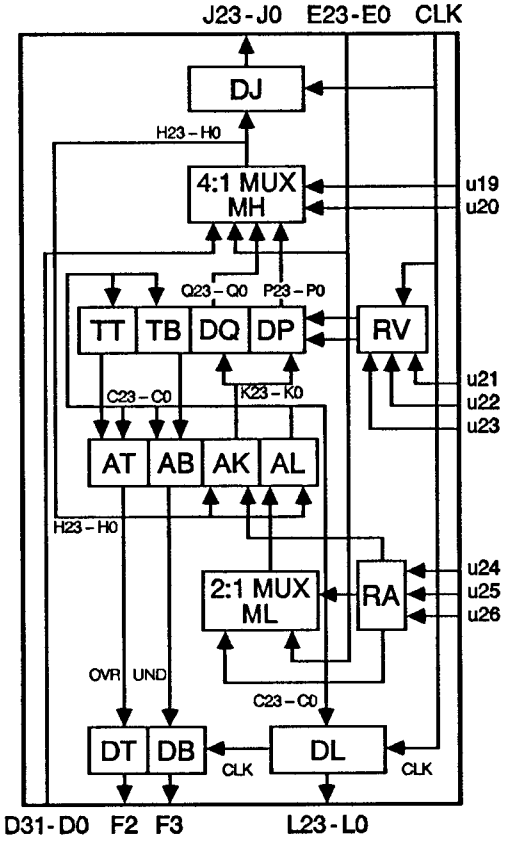


Figure 6. SPU Block Diagram.

The OVF box determines if a numeric overflow (NO) has occured. It is only set for SU and SL operations (see Table 5). For such operations:

If u11 = 1 - NO = SU32 xor SU31.

If u11 = 0 - NO = SU32.

The NO and ZO flags are latched and are available off chip as F1 and F0.

5. SPU Architecture

Figure 6 shows a simplified block diagram of the Stack Pointer Unit (SPU). It consists of two: stack pointer registers (DP & DQ), stack limit registers (TT & TB), adders (AP & AQ), comparitors (AT & AB) and multiplexers (MH & ML). It's operation is controlled by eight bits of the microcode (u26–u19), as described in Tables 6,7a and 7b.

The value on the E-bus, because of the microcode pipeline, is one from the previous SPU instruction. Data written onto the J-bus (from H) is buffered by DJ, so synchronicity is maintained.

The value on the H bus is determined by the 4:1 multiplexer MH: controlled by u20 and u19 as shown in Table 6. The value H is used as the base value for the stack RAM address (L) and the pointer registers input (K). The actual values offset by the adders is either determined by u26–u24, via the RA logic, or by the value on the E-bus: as shown in Table 7a.

Whether latches DP, DQ, TT or TB store new values is determined by u23–u21, via the RV logic, as shown in Table 7b.

Thus we can simultaneously point with any offset from pointers (DP or DQ), and either increment, decrement or leave the pointers unchanged. The only restriction being that both offsets (for K and C) must be in the same direction.

The two comparators AT and AB output a high, on F2 and F3, when the L value overshoots or undershoots the values stored in TT (t) and TB (b) respectively (i.e. b<L<=t). These two registers must be loaded with the stack limits b and t.

Having two stack pointers enables us to simulate two stacks: the P-stack and the Q-stack. Usually P will be initialised to RAM location 0 and fill up successively higher locations; while Q will be initialized to all 1's and fill up successively lower ram locations.

The external RAM will then function as a stack, through the address suplied on L23–L0 (=C23–C0 delayed one cycle).

With reference to Figure 4, stack data is written by setting u16 low and read by setting u16 high. Data read on D31–D0 passes through a transparent latch, TS (see Figure 4). TS passes data unhindered when u16 is high (read); but, when u16 goes low it isolates S and transmits the value corresponding to when u16 was last high.

Select Lines			Output of A-Adders	
u26	u25	u24	K	C
0	0	0	H+1	H+0
0	0	1	H+1	H+1
0	1	0	H+1	H+2
0	1	1	H+1	H−E
1	0	0	H−1	H−0
1	0	1	H−1	H−1
1	1	0	H−1	H−2
1	1	1	H−1	H−3

Table 7a. RA Logic.

Select Lines			Load Reg.
u23	u22	u21	
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	B
1	0	1	T
1	1	0	Q
1	1	1	P

Table 7b. RV Logic.

6. M-word Instructions

We are now in a position to see what type of instructions can be performed by the ASP. These instructions are called M-words, and are determined by m26–m0.

Tables 8a to 8h illustrate some of the possible M-words. Each word has the function of its seven fields (though the A field has been split into the A,D and E fields) and the effect on the P and Q stacks. The entries in the Tables are not exhaustive, there being many other useful combinations. However, the ones in Table 8 are sufficient to efficiently implement a stack based high level language (such as C).

The example of DUP (duplicate the TOS), will be employed to help explain Table 8. A duplicate requires that TOS be copied into NOS, the old NOS is copied into the next free RAM location and the stack pointer incremented so it continues to point to the next free location.

7.1 R-field.

As shown in Figure 5, bits u11–u0 of the M-word determine what the R field does. For a DUP, u10–u6 select LO and RO to be 0 and X respectively; while u2–u0 set R to equal XO. Thus, the net effect of the ALU is to set R = Y.

7.2 F-field.

The 4:1 multiplexer, shown in Figure 4, selects the input to the NOS, via u13 and u12. For DUP it selects the T input, which comes from the TOS. Thus, the new value of NOS will equal the old value of TOS.

7.3 X-field.

Figure 4 shows that bits u15–u14 determine the value on the X bus, via ME. For DUP it sets X equal to T, forcing X to equal the old value of TOS.

7.4 A-field.

Figure 4 shows that bits u18–u16 determine what goes onto the the A, D and E busses: via the tristate drivers and transparent latch TS. For DUP it leaves A tristate, sets D = Y (i.e. write NOS into the stack RAM) and sets E = T.

7.5 H-field.

Figure 6 shows that bits u20–u19 determine the value on the H bus. For DUP it passes the contents of VP, the P stack pointer, to H.

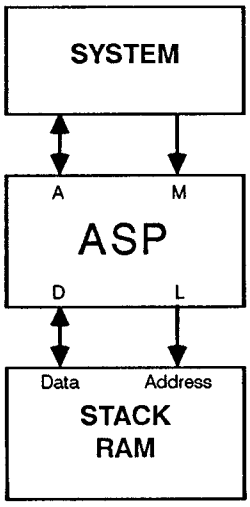


Figure 7. ASP System Interface.

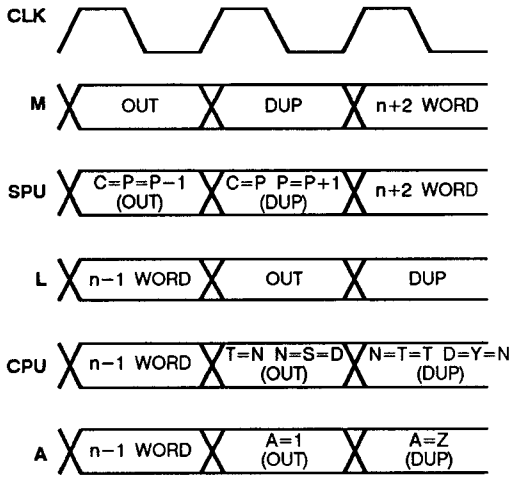


Table 9. ASP Timing Diagram.

7.6 L-field

Figure 6 shows that u26–u24 set the values on the L and K buses: via AL and AK. For a DUP, it selects L = H (so the data on the D bus is written into the next free location) and K = H+1 (so that P continues to address the next free location.

7.7 P-field.

Figure 6 shows that bits u23–u21 determine whether the value on the K bus is stored in the Q or P latch; and whether H is stored in the DU and DB latches. For DUP it stores the K, which equals the old P plus one, into VP.

7. ASP System Timing

Figure 7 shows the three basic subblocks in an ASP system: the ASP itself, the stack RAM and the rest of the system.

An actual computer system, capable of executing a HLL would require two ASP devices. One for address manipulation (ASP-A) and one for data manipulation (ASP-D).

The two chip system would contain a total of four stacks. The two data stacks (P-stack and Q-stack)) and the two address stacks (R-stack and S-stack). The two extra stacks, Q and S, made possible by the extra register in the SPU, allow extra flexibility in system design. Typical uses would be for loop counting, floating point numbers and HLLs requiring more than two stacks (eg LISP and Smalltalk).

The one chip system timings, are shown in Figure 10. Note particularly the delay between a change in M, and its effect on the D, L and A busses.

Table 9 illustrates the pipelined nature of the ASP. When a microcode word (say DUP) is put onto M, the SPU immediately starts its calculations. That is, C=P (next free stack RAM location) and P=P+1 (so the stack pointer continues to point to the next free location for the next instruction). At this time the CPU and stack RAM will still be doing the previous instruction.

During the next cycle, when a new M-WORD is placed on M, the address calculated by the SPU (and stored in DL) will be put onto the L bus. Also, the SPU will execute the DUP operation (TOS=NOS=old TOS and COS=next free RAM location=old NOS).

By having the stack address calculated in the previous cycle, it does not form part of the critical timing path: which in turn determines the maximum clock frequency.

'M' Word	CPU Field						SPU Field			P Stack	COM	M26 – M0 (Hex)	?
	R	F	X	A	D	E	H	L	P				
a) Stack Manipulation													
NOOP	X	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)ba		51DC1C4	P
SWASH	Y	T	*	Z	R	T	P	-1	P	dc(x)ba – ed(c)ab		5FDB344	P
DUP	X	T	T	Z	W	T	P	+0	P	dc(x)ba – cb(x)aa		0FCF1C4	P
DROP	Y	S	*	Z	R	T	P	-1	P	dc(x)ba – ed(c)cb		5FD2344	P
SWAP	Y	T	*	Z	R	T	P	-1	0	dc(x)ba – dc(c)ab		51D3344	P
OVER	Y	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ab		0FC3344	P
ROT1	X	T	S	Z	W	T	P	-1	0	dx(c)ba – db(c)ac		51CB1C4	P
PER1	X	S	T	Z	W	T	P	-1	0	dx(c)ba – db(c)ca		51CE1C4	P
NIP	X	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)ca		5FDE1C4	P
TIP	X	Y	T	Z	R	T	P	-2	P	dc(x)ba – ed(d)ba		6FDC1C4	P
b) P and Q Interstack Communication													
QTO	X	S	T	Z	W	T	Q	+0	Q	ed(c)ab – ed(c)cb	hg – ga	0D4E344	P
QFROM	X	Y	S	Z	R	T	Q	-1	Q	cb(x)ax – cb(g)ag	hg – ih	5D581C4	P
QCOPY	X	Y	S	Z	R	T	Q	-1	0	cb(x)ax – cb(g)ag	hg – hg	51581C4	P
c) Comparison Operations													
Ø<	(X–0)N	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g: a<0	51DC902	F
Ø>	(0–X)N	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g: a>0	51DC9D2	F
Ø=	(X–0)Z	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g: a=0	51DC9D3	P
>	(X–Y)N	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g: b>a	5FDEB12	F
<	(Y–X)N	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g: b<a	5FDE8D2	F
U>	(X–Y)NU	S	T	Z	R	T	P	-1	P	dc(x)ba – dc(c)bg	g: b>a	5FDE312	F
U<	(Y–X)NU	S	T	Z	R	T	P	-1	P	dc(x)ba – dc(c)bg	g: b<a	5FDE0D2	F
=	(X–Y)Z	S	T	Z	R	T	P	-1	P	dc(x)ba – dc(c)bg	g: b=a	5FDE8D3	P
d) Logical Operations													
AND	(X'+Y')'	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g=a.b	5FDE3C4	P
OR	(X'·Y')'	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g=a+b	5FDE3C5	P
XOR	X' xn Y	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g=a@b	5FDE0C6	P
NAND	(X·Y)'	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g=(a.b)'	5FDE005	P
NOR	(X+Y)'	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g=(a+b)'	5FDE004	P
XNOR	X xn Y	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g=a@b'	5FDE006	P
		S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg			
ANYOP	(X ? Y)	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g=a o b	5FDE???	P
NOT	(X+0)'	S	Y	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a'	51DC102	P

Tables 8a – 8d. M-Word Primitives.

'M' Word	CPU Field						SPU Field			P Stack	COM	M26 – M0 (Hex)	?
	R	F	X	A	D	E	H	L	P				
e) Arithmetic Operations													
+	Y+X	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g=b+a	5FDE800	P
-	Y+X'+1	S	T	Z	R	T	P	-1	P	dc(x)ba – ed(c)cg	g=b–a	5FDE8D0	P
1+	X+0+1	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a+1	51DC910	P
2+	X+2	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a+2	51DCC00	P
3+	X+2+1	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a+3	51DCC10	P
4+	X+4	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a+4	51DCD00	P
5+	X+4+1	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a+5	51DCD10	P
1-	X–2+1	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a–1	51DCE10	P
2-	X–2	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a–2	51DCE00	P
3-	X–4+1	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a–3	51DCF10	P
4-	X–4	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a–4	51DCF00	P
DAD1	Y+X	T	S	Z	R	T	P	-2	P	dc(x)ba – ed(d)ag	g=b+d	6FDB800	P
DAD2	Y+X+C	T	S	Z	R	T	P	-0	P	ed(x)ag – fe(c)gh	h=a+c+C	4FDB830	F
NEGATE	0+X'+1	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g= -a	51DC9D0	P
DNEG1	0+Y'+1	T	*	Z	R	T	P	-1	0	dc(x)ba – dc(c)ag	g= -b	51DFB50	P
DNEG2	0+Y'+C	T	*	Z	R	T	P	-1	0	dc(x)ba – dc(c)gh	h= -a+C	51DFB70	F
MUL	K+L	G	*	Z	R	T	P	-1	0	dc(x)ba – dc(c)gh	h,g=a*b	51D9AA0	–
U+	Y+X	S	T	Z	R	T	P	-1	P	dc(x)ba – dc(c)cg	g=b+a	5FDE000	F
U-	Y+X'+1	S	T	Z	R	T	P	-1	P	dc(x)ba – dc(c)cg	g+b–a	5FDE0D0	F
f) External Operations													
IN	X	T	E	Z	W	A	P	+0	P	dc(x)ba – cb(x)an	n=A	0FE31C4	P
IN4+	X+4	T	E	Z	W	A	P	+0	P	dc(x)ba – cb(x)an	n=A+4	0FE3500	P
OUT	Y	S	*	E	R	T	P	-1	P	dc(x)ba – ed(c)cb	A=a	5F9A344	P
COUT	X	Y	T	E	R	T	P	-1	0	dc(x)ba – dc(c)ba	A=a	519C1C4	P
OUT4+	X+4	Y	T	E	R	T	P	-1	0	dc(x)ba – dc(c)bg	A=a	519C500	P
INP*	X	Y	T	Z	R	A	E	-1	0	dc(x)ba – xy(y)ba	P=A+1	1EFC1C4	P
INQ*	X	Y	T	Z	R	A	E	-1	0	dc(x)ba – dc(c)ba	Q=A+1	1CFC1C4	P
INT*	X	Y	T	Z	R	A	E	-1	0	dc(x)ba – dc(c)ba	T=A	1AFC1C4	P
INB*	X	Y	T	Z	R	A	E	-1	0	dc(x)ba – dc(c)ba	B=A	18FC1C4	P
g) Control Operations													
BR	X	T	E	Z	R	A	P	-1	P	dc(x)ba – cb(x)ag	g=A	5FE31C4	P
BRZ(F)	X	Y	T	Z	R	A	P	-1	0	dc(x)ba – cb(x)ag	g=a+1(Z)	51DC110	P
BRZ(T)	X	T	E	Z	R	A	P	-1	P	dc(x)ba – cb(x)ag	g=A(Z)	5FE31C4	P

Tables 8e – 8g. More M-Word Primitives. (* = 2 cycles)

'M' Word	CPU Field						SPU Field			P Stack	COM	M26 – M0 (Hex)	?
	R	F	X	A	D	E	H	L	P				
h) Short Litrerals													
LIT5	0+4+(1)	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g=5	0FCFD50	P
LIT4	0+4	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g=4	0FCFD40	P
LIT3	0+2+(1)	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g=3	0FCFC50	P
LIT2	0+2	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g=2	0FCFC40	P
LIT1	0+(1)	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g=2	0FCF950	P
LIT0	0+0	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g=0	0FCF940	P
LIT1-	0−2+(1)	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g= -1	0FCFE50	P
LIT2-	0−2	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g= -2	0FCFE40	P
LIT3-	0−4+(1)	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g= -3	0FCFF50	P
LIT4-	0−4	T	*	Z	W	T	P	+0	P	dc(x)ba – cb(x)ag	g= -4	0FCFF40	P
i) Shift Operations (L/R=<->: 0/S/L=0,S32,S0 FILL: +/− This, last op)													
TWO*	X (L0+)	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a*2	51DC901	P
TWO/	X (RS+)	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a/2	51DC907	F
ROTL	X (LS−)	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a SL	51DC909	F
ROTR	X (RL−)	Y	T	Z	R	T	P	-1	0	dc(x)ba – dc(c)bg	g=a SR	51DC90F	P
ROTR2	Y (RL−)	S	*	Z	R	T	P	-1	P	dc(x)bx – dc(c)cg	g=b SR	5FD284F	P
j) Stack Pointer Manipulation													
SPTO*	X	Y	T	Z	R	T	E	+0	P	cb(x)an – cb(b)an	n → P	1EDC1C4	P
SQTO*	X	Y	T	Z	R	T	E	+0	Q	cb(x)an – cb(b)an	n → Q	1CDC1C4	P
STTO*	X	Y	T	Z	R	T	E	+0	T	cb(x)an – cb(b)an	n → T	1ADC1C4	P
SBTO*	X	Y	T	Z	R	T	E	+0	B	cb(x)an – cb(b)an	n → B	18DC1C4	P
P>	X	Y	E	Z	R	T	P	-1	0	cb(x)ax – cb(b)an	P → n	51D41C4	P
Q>	X	Y	E	Z	R	T	Q	-1	0	cb(x)ax – cb(b)an	Q → n	51541C4	P
TOSPIC*	X	Y	S	Z	R	T	P	+0	0	cb(x)an – cb(b)av	v=(n)	00D81C4	P
k) Relative Stack Manipulation													
PICK1	X	Y	T	Z	R	T	P	+0	0	dc[x]bn – dc[b]bn	[ ]=(P)	01CC1C4	
PICK2	X	Y	S	Z	RW	T	P	-E	0	dc[b]bn – dc(v)bv	v=(P−n)	31D81C4	
-PICK1	X	Y	T	Z	W	T	P	-E	0	ba(x)vn – ba(x)vn	(P−n)=v	31CC1C4	
AP1	X	Y	T	Z	R	A	P	+0	0	cb[x]aa – cb[a]aa	[ ]=(P)		
AP2	X	Y	S	Z	W	T	P	-E	0	cb[a]aa – cb(v)av	v=(P−A)		
-AP1	X	T	Y	Z		T	P	+0	P	cb(x)av – ba(x)vv	(P−A)=g		

Tables 8h – 8k. More M-Word Primitives. (*=2 Cycles)

Forth Words	T	1st M Word	2nd M Word	3rd M Word	4th M Word	5th M Word
2DROP	2	DROP	DROP			
ROT	2	NOOP	ROT1			
-ROT	2	SWAP	PER1			
D+	2	DAD1	DAD2			
DNEG	2	DNEG1	DNEG2			
>Q	2	SWAP	QTO			
Q>	2	DUP	QFROM			
Q@	2	DUP	QCOPY			
@	2	OUTL	IN			
!	2	OUTL	OUT			
ATOP	2	INP	INP			
ATOQ	2	INQ	INQ			
ATOT	2	INT	INT			
ATOB	2	INB	INB			
SP!	2	SPTO	SPTO			
SQ!	2	SQTO	SQTO			
ST!	2	STTO	STTO			
SB!	2	SBTO	SBTO			
ABSPICK	2	TOSPIC	TOSPIC			
PICK	2	PICK1	PICK2			
-PICK	4	NOOP	-PICK1	DROP	DROP	
APICK	3	AP1	AP2	DROP		
-APICK	5	DUP	-AP1	-PICK1	DROP	DROP

Table 10. Some One Chip Forth Words

	13	12	11	10	9	8	7	6	5	4	3	2	1	
A	A26	D30	D29	D27	D25	D22	D20	D17	D14	D12	D10	D09	D01	A
B	A24	A27	D31	D28	D26	D23	D19	D16	D13	D11	D08	D07	L23	B
C	A23	A25	A31	A30	D24	D21	D18	D15	D05	D06	D03	D00	L22	C
D	A21	A22	VDP	A29						D04	D02	L21	L20	D
E	A19	A20	A28								L17	L19	L18	E
F	A16	A17	A18								L14	L16	L15	F
G	A14	GND	A15								L12	VDD	L13	G
H	A12	A11	A13								L09	L10	L11	H
J	A09	A08	A10								CLK	L07	L08	J
K	A07	A06	A01	M18						F02	GNP	L05	L06	K
L	A05	A03	A00	M00	M17	M11	M10	M13	M16	F03	M24	L02	L04	L
M	A04	M01	M02	M03	M05	M06	M09	M15	M20	M23	M25	L00	L03	M
N	A02	F01	F00	M04	M07	M08	M09	M14	M19	M22	M21	M26	L01	N
	13	12	11	10	9	8	7	6	5	4	3	2	1	

A00 L11	A16 F13	D00 C02	D16 B06	L00 M02	L16 F02	M08 N08	M24 L03
A01 K11	A17 F12	D01 A01	D17 A06	L01 N01	L17 E03	M09 M07	M25 M03
A02 N13	A18 F11	D02 D03	D18 C07	L02 L02	L18 E01	M10 L07	M26 N02
A03 L12	A19 E13	D03 C03	D19 B07	L03 M01	L19 E02	M11 L08	F00 N11
A04 M13	A20 E12	D04 D04	D20 A07	L04 L01	L20 D01	M12 N07	F01 N12
A05 L13	A21 D13	D05 C05	D21 C08	L05 K02	L21 D02	M13 L06	F02 K04
A06 K12	A22 D12	D06 C04	D22 A08	L06 K01	L22 C01	M14 N06	F03 L04
A07 K13	A23 C13	D07 B02	D23 B08	L07 J02	L23 B01	M15 M06	CLK J03
A08 J12	A24 B13	D08 B03	D24 C09	L08 J01	M00 L10	M16 L05	VDD G02
A09 J13	A25 C12	D09 A02	D25 A09	L09 H03	M01 M12	M17 L09	VDP D11
A10 J11	A26 A13	D10 A03	D26 B09	L10 H02	M02 M11	M18 K10	GND G12
A11 H12	A27 B12	D11 B04	D27 A10	L11 H01	M03 M10	M19 N05	GDP K03
A12 H13	A28 E11	D12 A04	D28 B10	L12 H03	M04 N10	M20 M05	– –
A13 H11	A29 D10	D13 B05	D29 A11	L13 G01	M05 M09	M21 N03	– –
A14 G13	A30 C10	D14 A05	D30 A12	L14 F03	M06 M08	M22 N04	– –
A15 G11	A31 C11	D15 C06	D31 B11	L15 F01	M07 N09	M23 M04	– –

Appendix Spec-A. ASP Pin Connection Diagram (Pins Up) & Signals