AMIR CPU: World’s First and Only 32-bit Softcore Processor in Schematic on Freeware Platform

Dr. Muhammad Nasir Ibrahim\textsuperscript{1}, Namazi Azhari\textsuperscript{2}, Adam Baharum\textsuperscript{1}, Dr. Mariani Idroas\textsuperscript{1}, Uswhah Khairudin\textsuperscript{1}, Johari Kassim\textsuperscript{1}, Md Mohar Muhammad\textsuperscript{1}

\textsuperscript{1}Universiti Teknologi Malaysia, \\
\textsuperscript{2}Pahlawan Mikropemproses Sdn Bhd

\textbf{Abstract.} The developing world has always been a spectator and consumer participant in the ICT technology since the pioneering computing years of post second world war of world economic growth. The developed world led by the US, Europe, Japan, and later South Korea have surged far ahead in ICT being represented by their own creation of microprocessor platforms evolving into an exclusive club which they locked with licensing. Their early advantage of mostly trial and error processor designs have proven to be a market success that discouraged new aspiring contenders. New CAD tools have arrived along with the cloud computing model presenting new opportunities. Major ICT developments such as the Microsoft Catapult project in 2012, and the Intel purchase of Altera for USD16.7 billion in 2016 should be perceived as alert warnings. To us this is the perfect time to capitalize on the mistakes of the earlier processor designs to create a new and novel 32-bit processor for the rest of the free world to hop along the ICT revolution into the cloud and IoT applications on Freeware platform as active creative participants. These aspirations have to begin at the education level with an easy to learn 32-bit processor model.

\textit{Keywords: Freeware, Softcore, RISC/CISC, FPGA, HDL/VHDL/Verilog}

\section{1. Introduction}

We believe that for the developing countries to enter the playing field in ICT in order to create technological innovations of their own, it is imperative to have control over one platform that is a major building block of ICT: the microprocessor which is free from licensing beginning at the education level. ICT education tools have advanced so far as to be able to leapfrog the traditional methods of circuit wiring of breadboarding by supplanting with Freeware development tools and Intellectual-Property (IP) components. This freeware IP is the AMIR 32-bit softcore processor which is also multi-platform able to target all similarly capable devices from different vendors.

The processor was created in schematic so as to be able to reach a wide user-base of science and engineering professionals including occasional users such as scientists, students, and hobbyists. Unlike HDL (Hardware Description Language) design entry such as VHDL and Verilog which are favored by electronic design engineers which require steep learning curve, the schematic design entry requires virtually zero learning curve.
We define softcore processor as:

- Processor distribution that exists only in software, i.e., entity can be emailed as files in a folder
- On a board, without power there is no ready processor hardware; processor comes to life after a software image file of it is uploaded to an FPGA (Field-Programmable-Gate-Array).

We define AMIR freeware platform as:

- All required software tools; development environment, components library used are free from IP license as they can be legitimately downloaded from provider's website as well as freedom to make copies, and distribute
- An additional freedom is added specifically for the AMIR platform is that any AMIR derivative designs have the same Freeware rights with the rights as proprietary products without returning source codes. The last requirement is what differentiates between Open-Source.
- All this applies only in an FPGA target.

The basic software tools & components under the AMIR softcore freeware platform are listed below:

- Quartus II Web edition or any other freely downloadable FPGA tools
- Built-in libraries consisting primitive logic gates, 74hc series ICs, FPGA SRAM
- AMIR softcore processor component
- AMIR assembler
- GCC (GNU compiler)
- Editors using built-in Microsoft such as: Notepad or WordPad
- Editors under Ubuntu such as: gedit, LibreOffice
- Schematic+PCB CAD using KiCad in Ubuntu and Windows.

2. History of Freeware and Softcore Processor

An essentially “free software system” or “Freeware” built around the Unix platform was started by Richard Stallman who announced the GNU project in 1983. By 1990 all major components of GNU was completed except the kernel. Linus Torvalds happened to create the Linux kernel around this time, releasing it as freeware in 1992. Thus the GNU/Linux freeware platform was born and influenced the trajectory of computing within two decades. Microprocessors followed a course of gradual improvements beginning from 4-bit, 8-bit, 16-bit before hitting the mature 32-bit in which Linux were required. When Linus Torvalds started Linux in 1991 as a hobby he chose a readily available platform: a 32-bit i386 PC thus unknowingly chosen the bit-width of his free OS based on Unix in which Posix was based. This already set the data size that a new 21st century CPU design should begin. A 32-bit CPU can now take place jobs done by 16-bit, 8-bit, 4-bit easily and in the same footprint.

Xilinx introduced the first FPGA in 1985 with the XC2064 chip having 1200 programmable logic gates, and 64 CLBs (Configurable-Logic-Blocks) in a 2-micron process CMOS silicon. This was the first technology that enabled simple 8-bit softcore processors to be implemented in a number of this part. In 1991 Xilinx made a major improvement to the FPGA by including SRAM blocks in the new XC4000 chip. The SRAM blocks opened the possibility to create a single-chip softcore processor. The same year Philip Freedin created the first general purpose softcore CPU, the R16 or RISC4005 a 16-bit RISC processor with 16 registers running at 20 Mhz. From there onwards new softcore processors were being created every year including soft versions of hard processor architectures from Intel, ARM, and others.
Currently as of March 2018 there are more than a hundred softcore processors consisting of 4-bit, 8-bit, 16-bit, 32-bit, 64-bit, and variable-bit data path. Roughly 40 of them are 32-bit. Most are from the US and Europe. Some are freeware while some require licensing. All exist in high-level description language (HDL) with only one done in schematic by Jan Gray which is a 16-bit softcore processor. This AMIR is the first and only one in the world that is 32-bit done in schematic and available as a freeware component for all target platforms.

We list below some of the 32-bit softcore processors available as freeware:

- ZPU is a 32-bit stack-based from Norway created in 2008 by Zylin which has the copyright
- F32C is a 32-bit RISC V implementation subset from the University of Zagreb, Croatia
- F-CPU (Freedom CPU linked to YASEP) started in 1999, restarted in 2004 was once an ambitious effort which is still unfinished
- YASEP 16/32 bits; RTL in VHDL, & asm in Java, microcontroller subset: ready
- OpenRISC from a loose-knit organization called OpenCores in 2002 offered a free 32-bit processor IP whose architecture is similar to Sparc
- RISC V from Berkeley, a creation of the RISC inventor Prof David Patterson is a 32/64/128-bit softcore based on the pioneering quantitative study
- ZIP CPU is a 32-bit integer RISC processor created by Dan Gisselquist from Virginia US in 2015

3. Why Another 32-Bit Processor and Soft-Core

This section highlights some of the motivations for a new 32-bit processor architecture beginning with a softcore implementation. Selection of 32-bit as the data size covers 8-bit, and 16-bit applications for biggest market share presently, and the future especially for IoT devices. Processors have evolved to two main application groups: 32-bit and 64-bit. Currently PCs are slowly migrating to 64-bit which is the domain of servers. 32-bit now covers many application areas in embedded systems and user/work devices. Embedded applications in IoT, Broadband equipment, Robotics, Instrumentation which are dominated by ARM. User/Work applications in PC, Internet access, Smartphones which are dominated by both Intel and ARM. The technical advantages of 32-bit over 8/16-bit rests on the following four main points:

- Addressing; 16-bit addressing of 64kB is now too small
- Instructions; wider bit-fields preserve operands, larger immediate data and offset branch
- Data processing; minimum floating point data definition is 32-bit. Smaller width requires complex multi-byte operation
- Counters; necessary part of hardware and software housekeeping; 8-bit counters overflow easily, 16-bit inadequate

When we asked a number of instructors with regard to teaching the architecture of major 32-bit processors which inherently must include the assembly language, they are frustrated by the complexity of the bizarre ISA of the most popular platforms. For example, why the Intel x86 created 95 different instructions for branches? Not many professionals realized that the big vendors covered their shortcomings by saying there is no need for using assembly language as the C language is efficient enough. But in education, learning a processor architecture has to include the ISA, which is accessible only through the assembly language. Bob Colwell the chief architect for Intel Pentium P6: Pentium II/III, and Celeron, said in his interview "...it was only when I took an assembly language course that I began to understand what a computer architecture was about."
Case in point; the Intel IA-32 or the x86 ISA consists of some 95 assembly instructions for branches alone which present one strong argument why the Intel architecture discouraged assembly language education.

3.1. Intimidating repertoire of arcane and obtuse instructions and mnemonics just for branches
The IA32/x86 ISA has an excess of branch instructions which are more than a complete set of all the instructions in many microprocessors. Many are seldom used, hard to learn and remember.

Table 1. Listing of all 95 branch instructions in left and right tables.

| No | Opcode | Instruction | Description | No | Opcode | Instruction | Description |
|----|--------|-------------|-------------|----|--------|-------------|-------------|
| 1  | 77     | JA rel8     | Jump short if Above | 49 | OF 8F | JG rel32 | Jump near if greater |
| 2  | 73     | JAE rel8    | Jump short if Above or Equal | 50 | OF 8D | JGE rel16 | Jump near if greater or Equal |
| 3  | 72     | JB rel8     | Jump short if Below | 51 | OF 8D | JGE rel32 | Jump near if greater or Equal |
| 4  | 76     | JBE rel8    | Jump short if Below or Equal | 52 | OF 8C | JL rel16 | Jump near if Less |
| 5  | 72     | JC rel8     | Jump short if Carry | 53 | OF 8C | JL rel32 | Jump near if Less |
| 6  | E3     | JCEQXZ rel8 | Jump short if Cx =0 | 54 | OF 8E | JLE rel16 | Jump near if Less or Equal |
| 7  | E3     | JCECXZ rel8 | Jump short if CX =0 | 55 | OF 8E | JLE rel32 | Jump near if Less or Equal |
| 8  | E3     | JCEGESQ rel8 | Jump short if Cx =0 | 56 | OF 8E | JNA rel16 | Jump near if Not Above |
| 9  | 74     | JE rel8     | Jump short if Equal | 57 | OF 8E | JNA rel32 | Jump near if Not Above |
| 10 | 7F     | JG rel8     | Jump near if Greater | 58 | OF 82 | JNE rel16 | Jump near if Not Above or Equal |
| 11 | 7D     | JGE rel8    | Jump near if Greater or Equal | 59 | OF 82 | JNE rel32 | Jump near if Not Above or Equal |
| 12 | 7C     | JL rel8     | Jump near if Less | 60 | OF 83 | JNB rel16 | Jump near if Not Below |
| 13 | 7E     | JLE rel8    | Jump near if Less or Equal | 61 | OF 83 | JNB rel32 | Jump near if Not Below |
| 14 | 76     | JNA rel8    | Jump near if Not Above | 62 | OF 87 | JNB rel32 | Jump near if Not Below or Equal |
| 15 | 72     | JNAE rel8   | Jump near if Not Above or Equal | 63 | OF 87 | JNB rel32 | Jump near if Not Below or Equal |
| 16 | 73     | JNC rel8    | Jump near if Not Carry | 64 | OF 83 | JNC rel16 | Jump near if Not Carry |
| 17 | 77     | JNBE rel8   | Jump near if Not Below or Equal | 65 | OF 83 | JNC rel32 | Jump near if Not Carry |
| 18 | 73     | JNC rel8    | Jump near if Not Carry | 66 | OF 85 | JNE rel16 | Jump near if Not Equal |
| 19 | 75     | JNE rel8    | Jump near if Not Equal | 67 | OF 85 | JNE rel32 | Jump near if Not Equal |
| 20 | 7E     | JNG rel8    | Jump near if Greater | 68 | OF 8E | JNG rel16 | Jump near if Greater |
| 21 | 7C     | JNGE rel8   | Jump near if Greater or Equal | 69 | OF 8E | JNG rel32 | Jump near if Greater |
| 22 | 7D     | JNLE rel8   | Jump near if Less or Equal | 70 | OF 8C | JNLE rel16 | Jump near if Not Greater or Equal |
| 23 | 7F     | JNLE rel8   | Jump near if Less or Equal | 71 | OF 8C | JNLE rel32 | Jump near if Not Greater or Equal |
| 24 | 71     | JNO rel8    | Jump near if Not Overflow | 72 | OF 8D | JNL rel16 | Jump near if Not Less |
| 25 | 78     | JNP rel8    | Jump near if Not Parity | 73 | OF 8D | JNL rel32 | Jump near if Not Less |
| 26 | 79     | JNS rel8    | Jump near if Not Sign | 74 | OF 8F | JNL rel16 | Jump near if Not Less or equal |
| 27 | 75     | JNLE rel8   | Jump near if Not Zero | 75 | OF 8F | JNLE rel32 | Jump near if Not Less or equal |
| 28 | 70     | JNL rel8    | Jump near if Not Overflow | 76 | OF 81 | JNLE rel16 | Jump near if Not Overflow |
| 29 | 7A     | JP rel8     | Jump near if Parity | 77 | OF 81 | JNLE rel32 | Jump near if Not Overflow |
| 30 | 7A     | JPE rel8    | Jump near if Parity Even | 78 | OF 88 | JNP rel16 | Jump near if Parity |
| 31 | 7B     | JPO rel8    | Jump near if Parity Odd | 79 | OF 88 | JNP rel32 | Jump near if Parity |
| 32 | 78     | JS rel8     | Jump short if Sign | 80 | OF 89 | JNS rel16 | Jump near if Not Sign |
| 33 | 74     | JZ rel8     | Jump short if Zero | 81 | OF 89 | JNS rel32 | Jump near if Not Sign |
| 34 | OF 87  | JA rel16    | Jump near if Above | 82 | OF 85 | JNZ rel16 | Jump near if Not Zero |
| 35 | OF 87  | JA rel32    | Jump near if Above | 83 | OF 85 | JNZ rel32 | Jump near if Not Zero |
| 36 | OF 83  | JAE rel16   | Jump near if Above or Equal | 84 | OF 80 | JE rel16 | Jump near if Overflow |
| 37 | OF 83  | JAE rel32   | Jump near if Above or Equal | 85 | OF 80 | JE rel32 | Jump near if Overflow |
| 38 | OF 82  | JB rel16    | Jump near if Below | 86 | OF 8A | JP rel16 | Jump near if Parity |
| 39 | OF 82  | JB rel32    | Jump near if Below | 87 | OF 8A | JP rel32 | Jump near if Parity |
| 40 | OF 86  | JBE rel16   | Jump near if Below or Equal | 88 | OF 8A | JPE rel16 | Jump near if Parity Even |
| 41 | OF 86  | JBE rel32   | Jump near if Below or Equal | 89 | OF 8A | JPE rel32 | Jump near if Parity Even |
| 42 | OF 82  | JC rel16    | Jump near if Carry | 90 | OF 8B | JPO rel16 | Jump near if Parity Odd |
| 43 | OF 82  | JC rel32    | Jump near if Carry | 91 | OF 8B | JPO rel32 | Jump near if Parity Odd |
| 44 | OF 84  | JE rel16    | Jump near if Equal | 92 | OF 88 | JS rel16 | Jump near if Sign |
| 45 | OF 84  | JE rel32    | Jump near if Equal | 93 | OF 88 | JS rel32 | Jump near if Sign |
| 46 | OF 84  | JZ rel16    | Jump near if Zero | 94 | OF 84 | JZ rel16 | Jump near if Zero |
| 47 | OF 84  | JZ rel32    | Jump near if Zero | 95 | OF 84 | JZ rel32 | Jump near if Zero |
| 48 | OF 8F  | JG rel16    | Jump near if Greater | 96 | OF 8F | JG rel32 | Jump near if Greater |

The above 95 instructions are derived from the basic 32 different codes from the mnemonics below:

JO, JNO, JS, JNS, JE, JZ, JNE, JNZ, JB, JNAE, JC, JNB, JAE, JNC, JBE, JNA, JA, JNBE, JL, JNGE, JGE, JNL, JLE, JNG, JI, JNLE, JP, JPE, JNP, JPO, JCXZ, JECXZ.

Additionally there are also the LOOP (LZ, LNZ) and the JUMP (JP) instructions. Compare these with just five for integer operations for the AMIR CPU, using the eXecute instructions:
Table 2. Branch instructions for the AMIR.

| No | Opcode | Instruction | Description          |
|----|--------|-------------|----------------------|
| 1  | 1B     | X =         | Execute if Equal     |
| 2  | 1C     | X <         | Execute if Less      |
| 3  | 1D     | X >         | Execute if Greater   |
| 4  | 1E     | X <=        | Execute if Less or Equal |
| 5  | 1F     | X >=        | Execute if Greater or Equal |

There is no ambiguity as to what the above do as they center around ZERO result. If condition is true, execute the next instructions, if false jump to the 24-bit offset value. There is no near/far to contend with. There are legacy reasons for having the “short”, and “near” mostly centering around the mindset of having variable-byte instructions with 8-bit immediate field, and byte-size registers beginning from the progenitor of x86 architecture, the 16-bit 8086. The only good reason being that they help pipelines when they minimize branches taken.

3.1.1. CPU builds a platform
A platform creates Intellectual-Property that benefits a nation by supporting education with freeware standardized components for freedom to the developing world. Platform also creates an ecosystem of hardware and software industry.

3.1.2 There is no processor from our part of the world
All popular processors whether hard or softcore came from the developed world notably the US with the Intel x86, Europe with the ARM, and Japan from several companies such as Fujitsu, Hitachi, NEC, Mitsubishi, Toshiba, and others. None came from the developing nations.

3.1.3 Softcore/Reconfigurable/FPGA CPU is the future in education and industry
The time has come to lead the way in education of the future in response to Intel’s USD16.7 billion gamble in 2015\(^\text{12}\), and Microsoft’s server farms use of specialized FPGA processors beginning 2014 codenamed “Catapult” making it mainstream cloud equipment\(^\text{13}\). The gist of the purchase is about a whole way of learning and building electronics which today centers around a microprocessor. Learning digital electronics starts from logic gates; the basic building block of all complex digital circuits in all smart devices. From logic gates, future scientists and engineers learn combinational logic, then FSM (Finite-State-Machines), then functional logic blocks, then on to complex functions interconnecting them, finally leading to microprocessors. These steps can now be integrated onto just one small motherboard consisting of a softcore CPU, the ultimate digital electronics learning and designing environment in school right up to professional level with the same development tools. Even without a physical board, the free software development tools can simulate the design so as the environment can be made available to every computer desk in which a student normally works as freeware laboratory. With freeware CPU which is multi-targeted, our educators in higher learning and industry have a complete set of components on a uniform platform to impart know-how as well as target for commercialization, joining the promising R&D community working their way to the digital future of IoT, internet hardware, and cloud applications.

4. Notable Features of AMIR CPU
The AMIR CPU is not just another 32-bit processor as it has a few innovative schemes of its own. We present here a simple instruction format depicting its consistency in which a cursory programmer’s model could be inferred.
Figure 1. Instruction format bit-field of the AMIR

The opcode is always 8-bit. If no registers are specified, the open 24-bit field will be an ordinal of an offset value. Listed in the following are some of the notable differentiating features of the AMIR CPU architecture that maps to the simplified block diagram below in Figure 2.

Figure 2. Simplified AMIR CPU architecture block diagram

4.1. Orthogonal instruction set makes easy to learn which is ideal for education
Orthogonality is synonymous to consistency in a processor ISA (Instruction-Set-Architecture). The main criteria for instruction set orthogonality is in the register set, in which all can be used for both addressing and data without specialized or implied use except for rare occasions. This is possible when there are many general-purpose registers available typically sixteen or more. Another criteria is in instruction use where the same instructions can be reused for other operations without creating new specialized instructions and mnemonics, thus making them easier to learn. This is possible with three-register instruction format.

Thus the AMIR CPU was designed with 256 registers using a three-register instruction format, in one stroke lowers the learning curve.
4.2. Better and simple data transfers with separate banks for I/O

Data transfers from Load-Store occur some 30%\textsuperscript{14,15} which could be more when considering transfers between I/O and memory. Direct data transfers are possible only when the source and destination are in different banks, otherwise either temporary buffer or a DMA engine is required. Ideally data-transfer scheme in different banks should look like this:

\begin{center}
\includegraphics[width=0.5\textwidth]{three_bank_direct_data_transfer.png}
\end{center}

\textbf{Figure 3.} The three-bank direct data transfer architecture of CISC

The Intel x86 provided specialized I/O transfers with the IN and OUT instructions involving registers. The INS and OUTS involving “strings” and memory use implied registers DX, ES, DI which falls in the non-orthogonal instruction set category synonymous with inconsistency and hard to learn. The ARM has no explicit I/O instructions at all as it uses memory-mapped I/O. All processor architectures by default are memory-mapped I/O including Intel x86. Memory-mapped I/O is a lazy man’s scheme or “do nothing” I/O design effort. This decision gave rise to unnecessary problems later on, such as cache-coherency\textsuperscript{16}, and Gigabytes memory module-I/O partitioning. ARM was careless not to foresee this in 1983 when it was designed around memory modules of Kilobytes which was easy to partition among I/O peripherals. To see this problem it created, give a graduate student a project how to access I/O when all 32-bit memory addresses are used by one 4 GB DIMM which is now a common commodity PC part.

The AMIR CPU I/O scheme is revolutionary in that it encourages a change in mindset such that there is no need to think in terms of I/O transfer. All are data transfers from one bank to another bank. The instructions all use the same M (Move) mnemonic. Transfers between banks are differentiated by the characters @,# producing better orthogonality as tabled below.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Transfer operation & Instruction  \\
Register to Memory & m ra, @rb  \\
Memory to Register & m @ra, rb  \\
Register to I/O & m ra, #rb  \\
I/O to Register & m #ra, rb  \\
Memory to I/O & m @ra, #rb  \\
I/O to Memory & m #ra, @rb  \\
\hline
\end{tabular}
\caption{Data transfer instructions for the AMIR.}
\end{table}
4.3. Neither RISC nor CISC – Takes the best features, leave the worst

The AMIR CPU is neither RISC nor CISC. We took the best features of both and leave the worst. Though the AMIR CPU is not microcoded, there is nothing inherently wrong with microcoding technology apart from its inefficient use of transistors for large inputs. The future has to include some adaptation of microcoding because of its speed advantage due to one-level of logic, and adaptability to correct some errors after manufacturing. LUT (Look-Up-Table) uses the same technology as microcoding and is the basic technology used in FPGAs.

The main philosophy of RISC which translates into a few main rules is to create a simple set of instructions that execute fast with consistent instruction format which facilitates pipelining. Thus one of the main rules, in which RISC is often called a Load-Store processor is a questionable tenet which we believe to be an error. From this Load-Store rule, RISC has to adopt the memory-mapped I/O data transfer when its rule states no direct memory-I/O transfer is allowed. Thus the ideal scheme of Figure 3 cannot be realized, and instead reduced to this:

![Figure 4. The two-bank indirect data transfer between memory and I/O architecture of RISC](image-url)

The wisdom for memory-mapped is based on two: 1) All the powerful instruction set used on memory can be used for I/O devices as well, and 2) No complicated hardware needed for isolated I/O operations.

We contend both reasons are now incorrect, though they made sense in the eighties when memory modules were in the kilobytes or megabytes, and software directly controls I/O devices. Firstly, present memory modules came in Gigabytes that it is difficult if not impossible to partition memory space into I/O addresses. Secondly, current portable OS software discourages direct control of devices other than just transfer of data block to maintain device driver independence. In another important area where RISC breaks down is when SIMD is required in packet processing, multimedia, and cryptography which the internet heavily relies on. RISC and SIMD are two opposing ideas in that RISC leans on single-cycle instruction whilst SIMD is multi-cycle.

RISC simple load-store and pipelining concept could be traced to Seymour Cray in 1964 used in the CDC6600, later John Cocke from IBM continued on it in mid 70’s and implemented in the IBM801 RISC CPU in 1981. David Patterson of Berkeley gave the concept its name “RISC” and popularized it with the RISC-I processor in the same year. IBM has now returned to CISC with the latest generation of its mainframe 64-bit processor chip, created in 2000 known as the IBM z/Architecture based on CISC microcode technology which enabled inherent parallel processing on I/O channels. In 2015 IBM announced the z13 CISC microprocessor for their z13 mainframe computers which at the time was the world’s fastest microprocessor.
The best contribution of RISC are the three-operand instructions, and pipelining though both did not come from RISC which only adopted and popularized it. Both originated from old mainframe implementations. The three-operand instructions came from Honeywell 1800 circa 1974, and pipelining from IBM 7030 Stretch circa 1960.

The three-operand instructions made many two-operand instructions obsolete like; Compare, Bit-masking, Test, and Translation/Decoding.

4.4. Flat and linear 256-register bank

Register oriented architecture best maps higher level language to machine implementation, such that all major processors adopted it. The only argument is in the optimum number of registers. Parameters are passed via registers if they fit within the register bank. If not, then via the stack, and if a block of parameters is needed, then via memory pointer. The pioneering processor chips use accumulators due to fabrication constraints that it is now discarded. The Intel x86 still carried this legacy albeit with more spare registers totaling eight. The ARM started with sixteen per mode totaling 6 modes of bank-switched registers. Later processors settled with thirty two general purpose registers. Once selected, this is fixed into the architecture and cannot be easily changed without affecting compatibility.

The AMIR CPU implemented 256 registers using a 24-bit register field specifier in the opcode. This leaves an 8-bit instruction field. The math coprocessor and other hardware accelerators use this same 8-bit opcode specifier to access the private coprocessors’ register banks. Therefore theoretically the AMIR CPU places an absolute limit of registers using the 24-bit offset in the 32-bit instruction for coprocessors.

We believe this flat and linear 256-register bank is the best choice for microcontroller applications as it operates in supervisor mode in which during an interrupt, the processor sees all registers thus eliminating bank-switching. For user mode, the local register set can be chosen to be 16 or 32. Bank-switch register sets is now considered a mistake and new processors no longer implement it.

The desire for more registers have been validated by new (2010) ARM 64-bit architecture, the ARMv8 or A64 with 32 registers instead of 16. The A64 also eliminated the bank-switching modes altogether, admitting it as a mistake in its document.

4.5. Interrupt scheme that is fast and simple

In almost all processors there are two default interrupt inputs, usually the maskable interrupts (INT) and the non-maskable interrupts (NMI) as in the Intel x86. ARM has the normal maskable interrupt (IRQ) and the fast interrupt (FIQ). If there are more input sources than these, then an interrupt controller must be devised which normally features an interrupt pending stack and a priority scheme. The AMIR CPU implements a simple single interrupt scheme with an expandable eight input sources with default interrupt vectors that correspond to addresses of interrupt handlers in low memory. A priority encoder with the LSB (Least-Significant-Bit) having the highest priority, gives an output to the INT vector-port with a default I/O address =1 which is read by the CPU to match the vector table of interrupt handlers. The interrupt lines are also ORed together into a single INT FF (flip-flop) input which enables the next instruction fetch to be the hardware interrupt opcode. If the INT vector=1 which corresponds to memory address=1, the jump is made to the 32-bit address value found at this location which is the interrupt handler for this interrupt source. Once the interrupt vector is determined, the CPU outputs an INTA (Interrupt-Acknowledge) signal which is combined with the interrupt line address to disable the INT FF. The interrupt source can be a combination of both level-triggered or edge-triggered. Similar to the Intel scheme, there is only one jump required to start the interrupt handler.
4.6. **DMA eliminated with block-transfer instructions**

The AMIR CPU is about the only known processor architecture that made the elimination of the DMA possible. DMA is a block transfer between a memory bank and a block of data in another bank usually an I/O in a synchronous-burst data transfer. The AMIR CPU with the isolated memory and I/O banks can execute a synchronous-burst data transfer using the normal moves, \texttt{M @Ra,#Rb,Rc}. If \texttt{Rc} is not specified, the same opcode executes a data transfer of only one 32-bit word. If \texttt{Rc} is specified, the 32-bit data transfer is made synchronously to an I/O port until \texttt{Rc}=0. \texttt{Rc} is recommended to have an upper limit of 0xFFFF or 64 kilowords when the CPU clock runs at 100 MHz to keep execution time within 20 milliseconds or under 100 hz to be within the recommended multitasking latency period or a video refresh period.

5. **Mistakes in the Two Most Successful Architectures: Intel X86, Arm**

We selectively compare the notable features of the two most successful processor architectures today; Intel and ARM which happen to represent CISC and RISC respectively. The Intel x86 which AMD also produced as the only competitor accounts for virtually 100% of all PCs in the world including the Apple Macs. On the other front which is the smartphone, with the exit of Intel Atom in 2016, ARM controls virtually 100% of mobile processors. Hence it makes most sense to bring out the notable mistakes on these two architectures. Both are not new architectures as they were designed before 1985 when PCs were just being accepted as office and productivity tools, and the OS for the period was single-tasking floppy-disk based. Chip density was in the 1-micron and above that put a limit into the innovative processor ideas possible. That was the mindset that created the Intel x86 architecture which was 16-bit at birth, and the ARM which was a simple 32-bit at birth. The ARM which came out in 1985 was seven years later than the Intel 8086 and was much superior with more registers and advanced software support of protected mode. This same year too Intel came out with the 80386 with all the desirable support of software OS such as Unix from the upgrade of the 80286 in 1982. So the first ARM appeared in an already better software requirement of the period. Since both represented the pioneering evolution of CISC and RISC designs respectively, naturally we came to expect mistakes in the trial-and-error implementations. We bring out here the most notable mistakes in both architectures that we have learned.

5.1. **Mistakes in the Intel x86 32-bit architecture**

The progenitor of the venerable Intel x86 architecture was the 8086 16-bit processor which came out in 1978. Though it had evolved into the improved 16-bit 80286 in 1982, the 32-bit 80386 in 1985, the 80486 with built-in math coprocessor in 1989, the superscalar improved 32-bit Pentium P5 in 1993, and the first 64-bit x86 created by AMD came out in 2003. Despite their increased data sizes they remained downward binary compatible. We selected here the most notable mistakes.

5.1.1. **The variable bytes instruction set**

The x86 came from the CISC mindset with many instruction cycles from the variable bytes instruction set. Intel had to make this classic CISC into RISC and had to retrofit the instructions into a uniform instruction size which RISC had already adopted it from its first inception. The mistakes must be patched with extra logic and silicon to bring them to the next performance level.

It was estimated that 30% of transistors in the Pentium were spent for x86 legacy support and 40% in P6.\textsuperscript{23} Pentium front end logic had to compensate for non-uniform instruction size which affects cache lines, and segmented memory support by a translation stage needed for efficient decoding. This is a significant portion of the 5.5 million transistors in the first P6 version of the Pentium Pro in the mid nineties.\textsuperscript{23}
5.1.2. Segmentation
This may rank as the top complaint against the x86 processor which caused unnecessary brain twisting and arguably many software bugs in addressing calculations so much so that we bring to light one quote:

“Computer historians will very likely conclude that the IBM/Microsoft choice of the segmented-memory Intel x86 processor family over the linear address-space Motorola 680x0 family for the IBM PC is one of the greatest mistakes ever made in computer system design.” (1994) Nelson H. F. Beebe, Center for Scientific Computing, University of Utah.

This memory segmentation is redundant when memory paging in virtual memory does the same thing.

5.1.3. Two-operand instructions
A three-operand instruction typical of RISC makes many two-operand instructions redundant. With AMIR CPU’s three-operand instructions such as the SUB R0,R1,R2 the Intels’ CMP (compare), BT (bit-test), TEST, XLAT are unnecessary. Examples given below.

A Compare (CMP) instruction requires a corresponding Jump conditional (Jcc) instruction.

```
CMP EAX, EBX          ; subtract value in EBX register from EAX register
JE  (32-bit offset)   ; jump if equal an offset 32-bit value from PC (Program-Counter)
```

There are many different compare instructions and mnemonics apart from the jump instructions. A three-operand instruction uses the same SUB R0,R1,R2 for integer arithmetic for all zero comparisons. The specialized compare instructions are not needed.

Testing a register with a value which is equivalent to masking operation.

Intel created a specialized instruction, TEST EAX, 32-bit data. A three-operand instruction uses the same AND R0,R1,R2 for Logical operation.

Decoding or translation is a common operation in which a value is required to be converted to another code. Intel created a specialized XLAT (Translation) instruction just for this purpose. For example a binary value is needed to be displayed to a Common-Anode 7-segment display. This requires a Look-Up-Table (LUT) to be created.

Intel’s method is complex and for most occasions hard to remember, that is why Intel seems to encourage to visualize its ISA as a black hole to be abstracted by the C language.

The following example shows how to convert binary 9 to 7-segment Common-Anode in a table at location 0x1000 using Intel x86 ISA.

**Table 4.** Address locations of common-anode codes.

| Address | Common-Anode Code |
|---------|-------------------|
| 1000    | C0                |
| 1001    | F9                |
| 1002    | A4                |
| 1003    | B0                |
| 1004    | 99                |
| 1005    | 92                |
| 1006    | 82                |
| 1007    | F8                |
| 1008    | 80                |
| 1009    | 90                |
| 100A    | 08                |
Intel’s XLAT implied instruction which takes no operand uses the 8-bit AL register as the binary value or index register, to index into the DS:BX memory to retrieve the code, and put it back into the AL register. For 16-bit or 32-bit values, such as may be used in internet routing tables, Intel says nothing.

### Table 5. Registers used in the LUT conversion for the Intel x86 ISA.

| Registers | Value | Comments |
|-----------|-------|----------|
| EBX       | 0     | 32-bit segment address for DS register (Intel’s peculiar addressing) |
| DS        | 0x1000| starting address of Common-Anode codes |
| AL        | 9     | binary value for conversion/index-pointer to code in table |
| AL        | 0x90  | code in table, moved back into AL register |

After executing XLAT, the code value 0x90 pointed by DS:EBX is moved to AL.

By comparison, the AMIR CPU uses the same 3-register ADD instruction which can be used for 8-bit, 16-bit, and 32-bit values.

**ADD R11,R20,R21**

### Table 6. Registers used in the LUT conversion for the AMIR ISA.

| Registers | Value | Comments |
|-----------|-------|----------|
| R20       | 0x1000| starting address of Common-Anode codes |
| R11       | 9     | binary value for conversion |
| R21       | 0x1009| index-pointer to code in table, resulting from the add operation |

#### 5.1.4. Too few general purpose registers
The eight general purpose registers have proven to be inadequate as they came from the early popular minimalist implementation of legacy accumulator mindset. They are not strictly general purpose as each are defined for certain operations in implied instructions. The register set is a good example of non-orthogonal design approach which is difficult to learn. This is the reason why we see a lot of push-pop stack operations in an x86 (IA-32) assembly listing.

Realizing this shortcoming, the 64-bit Intel x86 (Intel 64) increased the number of general purpose registers from eight to sixteen.

In comparison, the new 64-bit ARM had increased its 16 registers to 32 and eliminating push-pop stack operations altogether indicating studies that register numbers have an effect on its necessity.

#### 5.2. Mistakes in the ARM 32-bit architecture
Many of the mistakes of the ARM architecture were already brought out under section 4 “Notable Features of AMIR CPU”. We reiterate here from another perspective.
The migration of the ARM architecture from 32-bit to 64-bit provides a good reference for CPU design because of its extensive past implementation studies of what was proven to be good and bad ideas or mistakes. What was discarded implied to be mistakes as the ARMv8 or A64 is a brand new architecture and not an upgrade of the 32-bit ARM or A32. Unlike the Intel 64 which is an extension of the 32-bit x86 (IA-32) and downward binary compatible, the A64 is not binary compatible as the two ISAs are completely different. The A64 can run the old A32 codes by switching states. There are two different processors 22 within a single chip which is not the same as a dual-core which is a network of two clones.

5.2.1 Load-Store only data transfer
RISC employs only load-store data transfer that goes through registers as the standardized 1-cycle instruction with pipelining. This enforces data transfers between a register and a memory location with an immediate value or a register pointer. This prevents other innovative ways for direct transfer of data without having to go through registers.

5.2.2 Pure RISC fails in SIMD, packet data transfer
RISC tried so hard to keep everything executing within the same period of time, essentially one instruction or more per cycle to maintain the pipeline in the mindset and period of the early 1980’s when single-tasking was the norm. When the internet exploded circa 1994 and with it multimedia, the constraint of single-cycle instruction must be broken at many places. Multimedia requires SIMD which is multicycle instruction, and internet with packet communication lines is efficient with block data transfers.

5.2.3 I/O same bank as memory
Since RISC allows data transfers only between register and memory, it follows that I/O is conveniently lumped together as memory locations at the same time achieving what was thought as an advantage to be able to control devices directly. Today data transfers occur most frequently and in blocks between memory and graphics adapter, and SATA hard disks. RISC of course does not use the load-store instructions for these but use DMA. RISC places another constraint in block devices, that is, it must use DMA and the need for complex DMA controller. But by just allowing direct memory-I/O transfer in different banks allowed DMA engines to be removed.

Also as brought in section 4.3, under “4.0 Notable Features of AMIR CPU”, having I/O in the same bank as memory can be considered a mistake.

5.2.4. Program-Counter (PC) part of register
Many RISC architectures other than ARM exposed the PC and Stack-Pointer (SP) by dedicating registers to them. This is done for flexibility when only the basic leaf procedure call instruction BL (Branch with Link) is needed to synthesize other complex recursive subroutine calls. In the ARM, for every recursive function call, requires a few instructions for synthesis. In the end, it converged to only one favored method that it is no longer an advantage to expose the PC. This is why the PC is put back to where it should belong; inaccessible to the programmer in the A64 architecture 22. It was noted in the ARM A64 documentation that it introduced complications for the design of compilers and complex pipelines.

5.2.5 Bank-switched register file
Bank-switched register file was popularized in RISC I design in which the 32-bit SUN Sparc architecture borrowed in its register windows. From David Patterson’s seminal 1982 RISC paper in which he attributed part of RISC I superior performance came from procedure call mechanism using what he called “overlapping register banks”. His benchmarks used selected C routines which did not consider multitasking. In fact nowhere in his 25-page long paper multitasking was mentioned. Currently, its not only multitasking that any processor design has to take into consideration as design
requirement, but preemptive multitasking. It is this preemptive multitasking that killed Intel’s much touted 1989 RISC processor the i80860. All the multi-execution units, separate caches, branch-prediction and other fancy logic could not help the Intel i860 when it performed poorly in context switching or exception handling when caches flushed easily with every interrupt 25.

The SUN Sparc has a complex scalable register files from a minimum of 40 to 520 with each register set partitioned into a 4-byte set of in, local, out, and global registers. The SUN Sparc v8 implemented eight sets of register windows for a total of 264 registers 26.

Patterson removed register windowing in his latest 2010 RISC V 27 in favor of a single 32-register file. The 32-bit ARM followed a similar course with six banks of register files, for a total of 96 registers.

The ARM 64-bit A64 removed these register banks in favor of just one 64-bit general purpose register file numbering 32. The reason given in the ARM documentation for removing register banks is that more than half of them were under utilized; implying an architectural mistake 22.

Current OS such as Ubuntu and Windows use preemptive multitasking such that there are many more daemons and background processes that could fit in the limited number of register windows and bank-switched register files. The scheme then became a new bottleneck when the processor spent much time doing housekeeping chores in proportion to useful work.

5.2.6. Interrupts using software table instead of direct vector
The external hardware interrupts IRQ and FIQ use the vectors 0x18 and 0x1C respectively no matter how many interrupt sources are tied. Another branch instructions are usually found at the two addresses typically ldr pc,pc,0x100 creating double jumps. Only then the actual jump addresses are retrieved from an index value. The software counterpart, the SWI instruction has a 24-bit ordinal which further requires instructions to find the actual jump address in the interrupt table.

This drawback was noted and corrected by the Atmel AT91 series ARM CPU by introducing its own proprietary Advanced-Interrupt-Controller (AIC) which routed the single vector into a hardware table of addresses. The ARM has its own interrupt controller, the ARM PrimeCell VIC (Vectored-Interrupt-Controller) that does essentially the same thing as the Atmel AIC.

The Intel is quite similar to the Atmel’s AIC in that it defaults to an interrupt vector table in memory from addresses 0 to 0xFF.

5.2.7 Conditional codes for branch instructions
Condition codes sometimes called predication used in the ARM instruction (and later added to the Intel Pentium) wastes 4 bits in the instruction field with little gain, a novelty in the 80’s but not now implemented in most new 32-bit CPU architectures. It is only useful in not stalling pipeline, and reduced code size but gives diminishing returns if the block of codes within a set flag condition needs instructions which modify the flag.

Later on it was made redundant with branch prediction hardware, and complicated the implementation of out-of-order execution. Thus it can be considered a mistake when the ARM A64 dropped most of them.

6. Why Schematic
Professional designers may find HDL (Hardware-Description-Language) the ideal method for complex designs needing collaborative partners. It certainly has its merits as demonstrated by the fact that all softcore processors except one were created by it. The main advantage after overcoming the learning curve is the ability to design at a functional level rather than from logic gates, once a description is completed and verified, it could be ported to all FPGA platforms.

The steep learning curve required of them may well be justified, but for these three categories of people; 1) beginner, 2) STEM (Science-Technology-Engineering-Mathematics) professionals such as teachers, scientists, and mathematicians, and 3) hobbyists, are more suited to schematic design entry as it is symbolic in nature and the same in any platforms with virtually zero learning curve.
There are two most popular HDL design synthesis: VHDL (Very-High-Speed-Integrated-Circuit) HDL, and Verilog. We limit our representation of HDL to Verilog because it resembles C and less verbose than VHDL, therefore easier to learn.

Three main grievances against Verilog; steep learning curve, design not self-documenting, cannot import into schematic or suitable block diagram. The free RTL viewer is not a substitute as the designer has no control over its output which cannot convey comprehensible graphic information. The RTL viewer is automatically drawn; take it or leave it, as there is no way to edit. Even if it could be edited, there is no guarantee it will look the same with different compilation. A graphical representation of any functioning work not just circuit design gives the designer a model for any creative mind to work on improvements. A similar analogy can be given in building a 3D model of a machine or a house both of which could be described in terms of (x,y,z) coordinates. Nobody could visualize coordinates, but a 3D solid rendition gives a whole picture in which the creative minds understand.

![Verilog code can be generated from this schematic](image1)

Verilog code cannot generate this schematic

![Both can generate this symbol, but this symbol can only appear in schematic](image2)

An example shown below in Figure 5 is to compare the 3-stage ring-counter in Verilog on the left and its schematic on the right. One glance at the schematic and anyone who understands logic gates will immediately know its function. As for the Verilog code one has to read all the text from top to bottom like reading a story. Both can generate the same symbol which can only appear in schematic. It is this symbol capability of schematic which makes it self-documenting. This symbol is also “alive” in that it is a hierarchical embodiment of schematic design as it may contain other symbols and another complete and complex digital electronic designs in either HDL such as Verilog or schematic. The symbol could also be simulated.

```verilog
module dff(q,d,c);
    output q;
    input d,c;
    reg q;
    initial
        q=1'b1;
    always @(posedge c)
        q=d;
endmodule

module dff1(q,d,clk);
    output q;
    input d,clk;
    reg q;
    initial
        q=1'b0;
    always @(posedge clk)
        q=d;
endmodule

module ring(q,clk);
    inout [2:0]q;
    input clk;
    dff u1(q[0],q[2],clk);
    dff1 u2(q[1],q[0],clk);
    dff1 u3(q[2],q[1],clk);
endmodule
```

Figure 5. Comparison of Ring-Counter design with Verilog on the left and schematic on the right
A schematic design space or “block design” as it is aptly named by Altera Quartus II is a hierarchical graphical representation of a block of digital functional ideas consisting of both Verilog and schematic synthesis which can contain schematic design within schematic design or block within block. Whereas a Verilog text space cannot contain any schematic design.

A final plus for schematic is that it can also be converted to Verilog text codes in Altera Quartus II by just clicking the menu “Create HDL Design File for Current File” which gives a choice of “VHDL or Verilog”. The converse is not possible; Verilog codes cannot create a schematic.

7. AMIR CPU as Electronic Components

The AMIR CPU consists of the AMIR CPU core and a separate schematic description for I/O ports which can create different components with different symbols for different applications. Shown below are three AMIR CPU manifestations as components with different number of I/O ports. These processors can operate in parallel within a single FPGA chip. The program memories reside internally in each device.

The simplest small AMIR CPU with a single input and output pair below in Figure 6, the AMIR-1 embodied in CPU4 can be used as a door-bell tone generator, or a remote-control infra-red switch. Remember that each component is a powerful 32-bit processor running at 100 Mhz. The more complex embodiments in CPU1, CPU2, and CPU3 can include separate I/O or FIFO buffers. All these processors have built-in program memory so each can run in parallel in an asymmetric multiprocessing.

Figure 6. The four variations of AMIR CPU with different I/O port configurations connected in a single schematic for parallel processing in an asymmetric multiprocessing.
8. Conclusion
We have presented some of the highlights of a new softcore processor from the developing world represented by an ASEAN country. Though the region may be late in producing a local processor as an answer to the monopolies of the Intel and ARM platforms, the advantages of benefitting from the mistakes of trial-and-error of earlier implementations to create a new design from scratch without having to declare whether it is a RISC or CISC but simply a better architecture is capitalized. The AMIR CPU is not just another processor but presents key improvements that promise better design schemes facilitating innovative creative ideas with elegant simplicity. The weaknesses of the Intel x86 and ARM architectures are motivation enough to create an alternative ICT hardware solution on a freeware platform for the benefit of the developing world without investing in a physical electronic laboratory except for the ubiquitous PCs which are now part of a normal education and work production tool.

Acknowledgements
We are indebted to the support of UTM (Universiti Teknologi Malaysia) ICC (Innovation and Commercialization Center) at Skudai, Johor Baru, Johor state of Malaysia, in the form of grants to produce a working prototype of the AMIR CPU. Gratitude also goes to KTD (Kolej Teknologi Darul-Naim Kelantan state of Malaysia) for facilitating the AMIR Lab in the technological college.

References
[1] Free-Software-Foundation 2017 Overview of the GNU System, https://www.gnu.org/gnu/gnu-history.html
[2] Stephen R. Walli, Software Systems, Inc. POSIX: A Case Study in a Successful Standard
[3] Linus Torvalds 1991 Newsgroups: comp.os.minix; Subject: Gcc-1.40 and a posix-question
[4] Xilinx datasheet for XC2064 1985
[5] Bradly Fawcett 1994 Xilinx Xcell15 Q4 Examining the process
[6] Wim Roelandts 1999 Xilinx Xcell32 Q2; 15 years of innovation, Major Xilinx milestones
[7] Jan Gray 1991 ReConFig 2014 Keynote; RISC4005 The first monolithic general purpose FPGA CPU
[8] Wikipedia 2018 Soft microprocessor,
[9] Wikipedia and OpenCore 2018 Softcore processors
[10] Intel-64 and IA-32, 2011 Architectures Software Developer’s Manual Volume 2A: Instruction Set Reference, A-M
[11] Bob Colwell Aug 2009 Oral history of Robert P. Colwell pp 8,26, Intel Fellow, Chief architect of Intel Pentium P6 (Pentium Pro, Pentium II, Pentium III, Celeron), Interviewed by Paul N. Edwards, Assoc. Prof., U of Michigan School of Information, Oregon
[12] Eugene Kim Dec 29 2015 Business Insider 3 reasons why Intel pulled the trigger on a massive $16.7 billion deal, its biggest ever
[13] Clayton Cotterell for Wired Sep 25 2016 Microsoft bets its future on a reprogrammable computer chip
[14] Jan Gray Circuit Cellar March 2000, Part-1 pp 27 Building a RISC System in an FPGA
[15] James R. Goodman, Wei-Chung Hsu 1986 U of Wisconsin-Madison US pp 375 IEEE On the use of registers vs cache to minimize memory traffic
[16] Andrew Tanenbaum 2009 Modern Operating System 3rd edition 5.1.3 Memory-Mapped I/O pp 333
[17] W. Warner Dec 22 2004 Great moments in microprocessor history- The history of the micro from the vacuum tube to today's dual-core multithreaded madness
[18] C.H Sequin and D.A. Patterson 1982 U of California, Berkeley, US Design and Implementation of RISC I;
[19] IBM News Room/News Release Jan 13 2015 Armonk, NY *IBM Launches z13 Mainframe -- Most Powerful and Secure System Ever Built*

[20] Honeywell 1800 product description 1974 pp 15 Honeywell Electronic Data Processing, Wellesley Hills, Massachusetts, US.

[21] G.A. Blaauw and F.P. Brooks Jr. 1997 *Computer Architecture: Concepts and Evolution* section 13.3 *IBM Stretch*. Reading, MA: Addison Wesley

[22] ARM Cortex-A Series version 1.0 2015 *Programmer’s Guide for ARMv8-A* Chap-4 ARMv8 Registers pp 4-14

[23] Jon Stokes 2007 *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture* William Pollock (Pub.), Ch 5 *The Intel Pentium and Pentium Pro* pp 92, pp107, No Starch Press, Inc, San Francisco, USA.

[24] Md Nasir, N Azhari, A Baharum 2017 Pahlawan Mikropemproses Sdn Bhd, Kota Bharu Kelantan, Malaysia *AMIR-I 32-bit Microcontroller Instruction Set Architecture*

[25] Paul DeMone Jan 25 2000 real world technologies *Intel’s History Lesson* pp 2 [https://www.realworldtech.com/intel-history-lesson/](https://www.realworldtech.com/intel-history-lesson/)

[26] SPARC International Inc. Menlo Park California US 1992 *The SPARC Architecture Manual Version 8* pp 9, 24

[27] David A. Patterson et al 2014 EE & Computer Sciences University of California Berkeley US *The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.0*