Pep/10

J. Stanley Warford
Matthew McRaven

March 19, 2021

Here are the differences between Pep/10 and Pep/9 along with a rationale for each change.

1. `STOP` replaced by `RET`

   The `STOP` instruction is no longer in the instruction set. Instead, the operating system now calls the C `main()` function with the system return value preset to 0. The translation more closely matches the terminating C statement

   ```
   return 0;
   ```

   The symbolic debugger of the Pep/10 IDE now shows the run-time stack from the OS call with two cells – `retAddr` and `retVal`. If students terminate their programs with `RET` the return value will be 0 because that is the preset return value, and control is returned to the simulator the same way a `STOP` instruction does in Pep/9. However, if they modify the value before the return, the OS issues an error message with an echo of the error number.

   The operating system has a new dispatcher component as the interface between the OS and the application. This interface is more realistic of the way C works and reenforces the concept that the operating system calls the application, and the application returns control to the operating system.

2. Memory-mapped shutdown port

   Pep/9 introduced the concept of memory-mapped I/O ports. In another step toward hardware realism, and to have a mechanism for terminating a simulation, Pep/10 has a memory-mapped shutdown port. If any value at all is written to the port the simulation is terminated and control is returned to the IDE.

   Students first learn how to program in machine language at the ISA3 level without the assistance of the operating system. They learn how to store a byte to the output port with direct addressing to output an ASCII character. In Pep/10, they simply store a byte to the shutdown port with direct addressing to terminate their programs.

   There are two benefits to this feature. First, students do not need to learn a new `STOP` instruction to terminate their machine language programs. But more importantly, they learn the utility of memory-mapped device registers with this rudimentary example.

3. Trap instructions replaced by system calls

   Pep/9 has five trap instructions – `NOP`, `DECI`, `DECO`, `HEXO`, and `STRO`. Pep/10 replaces them with two system calls – `SCALL` for system call and `USCALL` for unary system call. For example, `DECI` is no longer an instruction mnemonic as it is in Pep/9, but a symbol exported from the operating system. This decimal input instruction in Pep/9

   ```
   DECI    num,d
   ```

   becomes the following system call in Pep/10

   ```
   LDWT    DECI,i
   SCALL   num,d
   ```

   Pep/10 has a new Trap register visible at level ISA3 used by the system call instructions. `LDWT` is the load word trap instruction, which loads the entry point address of the `DECI` code in the operating system.

   In Pep/9, the use of the trap instructions so closely parallels the use of the native ISA instructions that beginning students frequently do not even realize they are system calls. Pep/10 has the pedagogic advantage of making system calls explicit, and is more realistic. From a system design perspective, the trap instructions no longer occupy the opcode space which opens the possibility for new instructions. The new design also does not constrain the number or variety of system calls.

4. Assembler macro facility

   In yet another step toward industry standard practice, Pep/10 introduces assembler macro expansions. The IDE provides cononical standard macros for all the system calls. With the supplied macros, the above `DECI` example becomes simply

   ```
   @DECI   num,d
   ```

   where `@DECI` is now the macro name. The generated program listing shows the macro source instruction and its expansion. The convenient `CHARI` and `CHARO` instructions from Pep/8 and earlier are now back as macros `@CHARI` and `@CHARO`. This is a pedagogical improvement over Pep/9, because now character I/O is programmed exactly like decimal I/O. With dynamic allocation, students no longer need to copy/paste the code for `malloc()` at the end of their source because `@MALLOC` is a supplied macro.

   It is possible for students to write their own macros with the Pep/10 IDE, which contains documentation for how to do so. However, writing macros is outside the scope of this text.

5. Easier modification of OS

   In Pep/9, to write a new trap instruction the IDE requires you to redefine one of the existing mnemonics. Now that trap instructions are replaced with system calls there is no longer such a requirement. The Pep/10 assembler uses the new `.SCALL` and `.USCALL` directives to automatically create system call macros. Operating system programmers may use the `.EXPORT` directive to control which symbols are available from application code. By combining these facilities, students can write any number of system calls unconstrained by the opcode space.

   The full declaration of a system call in the operating system is as follows:

   ```
           .EXPORT mymacro
           .USCALL mymacro
   mymacro: RET
   ```

6. Explicitly declared IO ports

   Pep/10 introduces two new directives: `.INPUT` and `.OUTPUT`. These directives indicate to the simulation and student that associated symbols correspond to user-accessible devices. Operating system programmers will receive warnings from the Pep/10 assembler if they have disconnected IO devices, preventing silent IO failures caused by misspelled symbols. An IO directive does not automatically mark the symbol as `EXPORT`'ed, maintaining parity with system call declarations.

   The full declaration for a memory-mapped IO port is as follows:

   ```
           .EXPORT port
           .OUTPUT port
   port: .BLOCK  1
   ```

7. New disk input port

   In another bid towards a more realistic system model, the loader now takes its input from a disk input port.

8. New instruction `XORr`

   Deletion of the trap instructions from Pep/9 opens up the opcode space for new instructions. Finally, the exclusive OR instruction `XORr` is a native ISA instruction.

9. Improved instruction set

   The instruction set is now more representative of real ISA instruction sets. Neither the `STOP` instruction nor the specialized trap mnemonics of Pep/9 are in actual ISA sets. System calls are also typical. Both Pep/9 and Pep/10 have 40 ISA instructions, but the Pep/10 set is more regular. The opcode space is cleaner, with all the unary instructions followed by all the branch instructions with two addressing modes followed by the remaining instructions with eight addressing modes.

10. Improved dot command set

    The `.ADDRSS` pseudo-op is deleted and replaced by the existing `.WORD` pseudo-op. They both perform the same function, inserting a two-byte value into the object code. Previously, `.WORD` was limited to a constant arguments, and `.ADDRSS` was limited to symbol arguments, which are addresses. Now `.WORD` accepts both constant and symbol arguments.

11. Modified CPU

    At the ISA level, the only difference in the CPU is the new 16-bit Trap register, which is necessary to make system calls. At the LG1 level, the Pep/10 data section for the one-byte bus is identical to that of Pep/9, again with the only difference being the labeling of the Trap register. The two-byte bus model is deleted altogether in favor of expanded coverage of an industry standard RISC chip.

12. New RISC-V microarchitecture

    The MIPS processor in previous editions is replaced by the open-source hardware standard RISC-V, which is presented throughout the text. The sidebars that describe the x86 architecture in the previous edition now describe RISC-V. The previous edition contains microcode programming problems for the one-byte bus and two-byte bus versions of the Pep/9 CPU data section. MIPS assembly language exercises are limited to paper. This edition replaces the MIPS paper exercises with RISC-V assembly language programming problems. Students implement components of the Pep/10 computer at the Mc2 level with microcode as before. Now they also implement components of the Pep/10 computer at the ISA level with RISC-V assembly language code.

13. Improved register transfer language

The register transfer language syntax more closely follows industry practice. Pep/10, which is a big endian machine, indexes the most significant bit on the left as 0, and increases from left to right. RISC-V, which is a little endian machine, indexes the least significant bit on the right as 0, and increases from right to left. Bit ranges now use the standard syntax with square brackets and a colon.
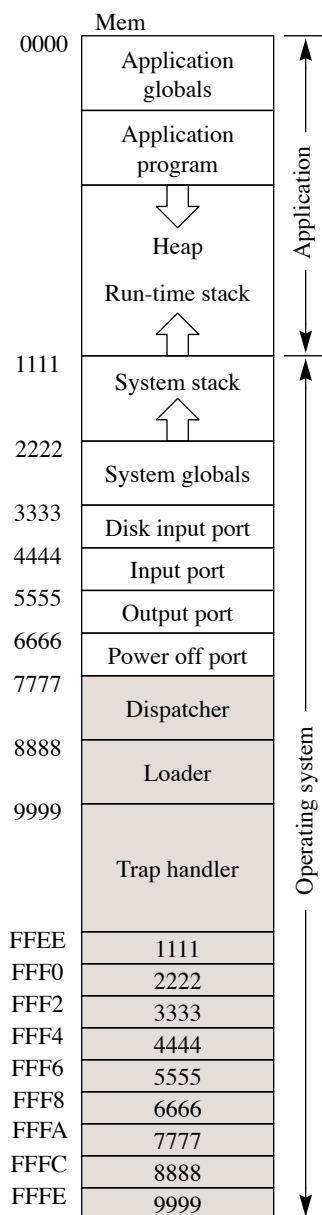
14. New software support

The suite of software for Pep/10 includes an assembler/simulator at the ISA level and a micro-assembler and simulator at the Mc2 level as before. The micro-assembler has a unit pre- and post-condition facility for students to test their programs. The suite now includes a RISC-V assembler/simulator with the same pre- and post-condition facility. The application maps the Pep/10 ISA registers onto a subset of the RISC-V registers for students to simulate Pep/10 at the ISA level. They experience the difference between CISC and RISC using RISC-V as a form of microcode to simulate Pep/10.

Instruction Specifier	Mnemonic	Instruction	Addressing Mode	Status Bits
0000 0000	RET	Return from CALL	U	
0000 0001	SRET	Return from system CALL	U	
0000 0010	MOVSPA	Move SP to A	U	
0000 0011	MOVASP	Move A to SP	U	
0000 0100	MOVFLGA	Move NZVC flags to A[12 : 15]	U	
0000 0101	MOVAFLG	Move A[12 : 15] to NZVC flags	U	NZVC
0000 0110	MOVTA	Move T to A	U	
0000 0111	USCALL	Unary system call	U	
0000 1000	NOP	No operation	U	
0001 000r	NOTr	Bitwise invert r	U	NZ
0001 001r	NEGr	Negate r	U	NZV
0001 010r	ASLr	Arithmetic shift left r	U	NZVC
0001 011r	ASRr	Arithmetic shift right r	U	NZC
0001 100r	ROLr	Rotate left r	U	C
0001 101r	RORr	Rotate right r	U	C
0001 110a	BR	Branch unconditional	i, x	
0001 111a	BRLE	Branch if less than or equal to	i, x	
0010 000a	BRLT	Branch if less than	i, x	
0010 001a	BREQ	Branch if equal to	i, x	
0010 010a	BRNE	Branch if not equal to	i, x	
0010 011a	BRGE	Branch if greater than or equal to	i, x	
0010 100a	BRGT	Branch if greater than	i, x	
0010 101a	BRV	Branch if V	i, x	
0010 110a	BRC	Branch if C	i, x	
0010 111a	CALL	Call subroutine	i, x	
0011 0aaa	SCALL	System call	i, d, n, s, sf, x, sx, sfx	
0011 1aaa	LDWT	Load word T from memory	i	
0100 raaa	LDWr	Load word r from memory	i, d, n, s, sf, x, sx, sfx	NZ
0101 raaa	LDBr	Load byte r[8 : 15] from memory	i, d, n, s, sf, x, sx, sfx	NZ
0110 raaa	STWr	Store word r to memory	d, n, s, sf, x, sx, sfx	
0111 raaa	STBr	Store byte r[8 : 15] to memory	d, n, s, sf, x, sx, sfx	
1000 raaa	CPWr	Compare word to r	i, d, n, s, sf, x, sx, sfx	NZVC
1001 raaa	CPBr	Compare byte to r[8 : 15]	i, d, n, s, sf, x, sx, sfx	NZVC
1010 raaa	ADDr	Add to r	i, d, n, s, sf, x, sx, sfx	NZVC
1011 raaa	SUBr	Subtract from r	i, d, n, s, sf, x, sx, sfx	NZVC
1100 raaa	ANDr	Bitwise AND to r	i, d, n, s, sf, x, sx, sfx	NZ
1101 raaa	ORr	Bitwise OR to r	i, d, n, s, sf, x, sx, sfx	NZ
1110 raaa	XORr	Bitwise XOR to r	i, d, n, s, sf, x, sx, sfx	NZ
1111 0aaa	ADDSP	Add to SP	i, d, n, s, sf, x, sx, sfx	NZVC
1111 1aaa	SUBSP	Subtract from SP	i, d, n, s, sf, x, sx, sfx	NZVC

Instruction	Register transfer language specification
RET	$PC \leftarrow Mem[SP]$; $SP \leftarrow SP + 2$
SRET	$NZVC \leftarrow Mem[SP][4:7]$; $A \leftarrow Mem[SP+1]$; $X \leftarrow Mem[SP+3]$; $PC \leftarrow Mem[SP+5]$; $SP \leftarrow Mem[SP+7]$
MOVSPA	$A \leftarrow SP$
MOVASP	$SP \leftarrow A$
MOVFLGA	$A[8:11] \leftarrow 0$, $A[12:15] \leftarrow NZVC$
MOVAFLG	$NZVC \leftarrow A[12:15]$
MOVTA	$A \leftarrow T$
USCALL	$Y \leftarrow Mem[FFF0]$; $Mem[Y-2] \leftarrow SP$; $Mem[Y-4] \leftarrow PC$; $Mem[Y-6] \leftarrow X$; $Mem[Y-8] \leftarrow A$; $Mem[Y-9][4:7] \leftarrow NZVC$; $SP \leftarrow Y-9$; $PC \leftarrow Mem[FFFE]$
NOP	{*No operation*}
NOTr	$r \leftarrow \neg r$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$
NEGr	$r \leftarrow -r$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$, $V \leftarrow$ {*overflow*}
ASLr	$C \leftarrow r[0]$, $r[0:14] \leftarrow r[1:15]$, $r[15] \leftarrow 0$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$, $V \leftarrow$ {*overflow*}
ASRr	$C \leftarrow r[15]$, $r[1:15] \leftarrow r[0:14]$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$
ROLr	$C \leftarrow r[0]$, $r[0:14] \leftarrow r[1:15]$, $r[15] \leftarrow C$
RORr	$C \leftarrow r[15]$, $r[1:15] \leftarrow r[0:14]$, $r[0] \leftarrow C$
BR	$PC \leftarrow Oprnd$
BRLE	$N = 1 \vee Z = 1 \Rightarrow PC \leftarrow Oprnd$
BRLT	$N = 1 \Rightarrow PC \leftarrow Oprnd$
BREQ	$Z = 1 \Rightarrow PC \leftarrow Oprnd$
BRNE	$Z = 0 \Rightarrow PC \leftarrow Oprnd$
BRGE	$N = 0 \Rightarrow PC \leftarrow Oprnd$
BRGT	$N = 0 \wedge Z = 0 \Rightarrow PC \leftarrow Oprnd$
BRV	$V = 1 \Rightarrow PC \leftarrow Oprnd$
BRC	$C = 1 \Rightarrow PC \leftarrow Oprnd$
CALL	$SP \leftarrow SP - 2$; $Mem[SP] \leftarrow PC$; $PC \leftarrow Oprnd$
SCALL	$Y \leftarrow Mem[FFF0]$; $Mem[Y-1] \leftarrow IR[0:7]$; $Mem[Y-3] \leftarrow SP$; $Mem[Y-5] \leftarrow PC$; $Mem[Y-7] \leftarrow X$; $Mem[Y-9] \leftarrow A$; $Mem[Y-10][4:7] \leftarrow NZVC$; $SP \leftarrow Y-10$; $PC \leftarrow Mem[FFFE]$
LDWT	$T \leftarrow Oprnd$
LDWr	$r \leftarrow Oprnd$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$
LDBr	$r[8:15] \leftarrow$ byte $Oprnd$; $N \leftarrow 0$, $Z \leftarrow r[8:15] = 0$
STWr	$Oprnd \leftarrow r$
STBr	byte $Oprnd \leftarrow r[8:15]$
CPWr	$Y \leftarrow r - Oprnd$; $N \leftarrow Y < 0$, $Z \leftarrow Y = 0$, $V \leftarrow$ {*overflow*} , $C \leftarrow$ {*carry*} ; $N \leftarrow N \oplus V$
CPBr	$Y \leftarrow r[8:15] -$ byte $Oprnd$; $N \leftarrow Y < 0$, $Z \leftarrow Y = 0$, $V \leftarrow 0$, $C \leftarrow 0$
ADDr	$r \leftarrow r + Oprnd$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$, $V \leftarrow$ {*overflow*} , $C \leftarrow$ {*carry*}
SUBr	$r \leftarrow r - Oprnd$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$, $V \leftarrow$ {*overflow*} , $C \leftarrow$ {*carry*}
ANDr	$r \leftarrow r \wedge Oprnd$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$
ORr	$r \leftarrow r \vee Oprnd$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$
XORr	$r \leftarrow r \oplus Oprnd$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$
ADDSP	$SP \leftarrow SP + Oprnd$
SUBSP	$SP \leftarrow SP - Oprnd$

Here is the memory map of the Pep/10 system. The shaded portion is ROM. Compared to the Pep/9 memory map, this map has several additional components – a disk input port at address 3333, a power off port at address 6666, and a dispatcher at address 7777. Pep/9 has six machine vectors from FFF4 to FFFE. Because of the new components, Pep/10 has nnine machine vectors at addresses FFEE to FFFE.

	Mem	
0000	Application globals	▲
	Application program	
	⇩	Application
	Heap	
	Run-time stack	
	⇧	▼
1111	System stack	▲
	⇧	
2222	System globals	
3333	Disk input port	
4444	Input port	
5555	Output port	
6666	Power off port	
7777	Dispatcher	
8888	Loader	Operating system
9999	Trap handler	
FFEE	1111	
FFF0	2222	
FFF2	3333	
FFF4	4444	
FFF6	5555	
FFF8	6666	
FFFA	7777	
FFFC	8888	
FFFE	9999	▼

Here is the data section of the Pep/10 CPU. The only difference from P/9 is the addition of the trap register T and the labels for the scratch registers R0 – R4 and the constant registers C0 – C4.