2 An overview of the Machine and Run-Time Environment

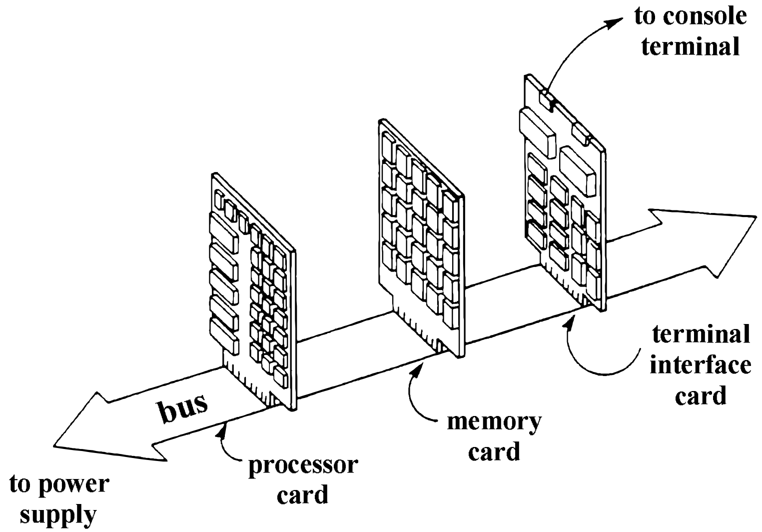
2.1 The Machine

- The machine discussed here is a **Digital Equipment Corporation LSI 11/2** 16-bit microprocessor. This is a microcomputer version of the PDP 11.
- We will discuss:

processor	communication devices	mechanisms such as
memory	asynchronous comm. devices	– stack
architecture	disk storage devices	– vectored interrupts
		– device addressing

2.1.1 Physical Organization of the LSI 11/2

- The LSI 11/2 is constructed from **printed circuit boards** which are slotted into the **sockets of a backplane**.



- Sockets are wired together to form a **bus** (the Q-bus) consisting of **power lines** (attached to all board) and **lines linking up the sockets for inter-board communication**: signals travel to the board on one contact and away on another. One board contains the **11/2 processor** itself (at the beginning of the bus), other boards contain memory or device interfaces for I/O devices.

Communication between boards

- A board communicates with another board by **passing signals across the bus**. E.g., when the processor board needs to **write into memory**, it places **address + data** on the bus for the memory board to retrieve and store.
- Memory is **logically contiguous**: each board contains switches (hardwired jumpers) that can be changed – it is possible to configure **two identical memory boars** so that one responds to **low memory addresses** and the other to **high addresses**. Similarly, I/O devices interface boards contain such switches to represent two distinct devices.

Communication between boards

- The physical order of boards along the Q-bus determines their **priority** (this is used later): Signals enter a board on one connector and leave it on another one, so the board can decide whether to **intercept the signal** or **pass it on down the bus**.
- Example:
  - Two boards are waiting for service.
  - The CPU sends a **grant signal** down the bus.
  - It **passes from board to board** until it reaches a board ready for service.
  - The first board with a device waiting responds to the signal **without passing it on**.
  - Boards further down the bus **wait** until the CPU finishes servicing the first board and reissue the grant signal.

2.1.2 Logical Organization of the LSI 11/2

- The Operating system is concerned with the logical organization of the machine, not its physical organization.

2.1.3 The Registers of the LSI 11/2

- The LSI 11/2 processor contains eight 16-bit **registers** (called $R_0, \ldots, R_5, SP$ and PC) and one 16-bit **processor status word** (PS):

Notation	Register	Use	
R_0	0	general purpose	
R_1	1	general purpose	
R_2	2	general purpose	
R_3	3	general purpose	
R_4	4	previous display	$\longleftarrow$ points to the previous display, i.e. to the calling procedure's frame
R_5	5	current display	$\longleftarrow$ points to the current display, i.e. to the currently active called procedure
PC	7	program counter	
PS	status	processor status	

2.1.4 The Address Space

- Memory is divided into 8-bit quantities called **bytes** – also called characters – with the byte being the **smallest addressable unit**.
- Most instructions operate on two bytes (a **word**), the adressed byte and the next higher byte:

address of byte:	addressed byte	$\longleftarrow$ address of the word
address of byte +1:	next higher byte	

- The 11/2 adresses are formulated in **octal**, there are $64k$ Bytes of memory (with $k = 1024$): 2^{16} Bytes $=$ 2^{15} Words.

The Address Space

$000\,0000_8$:	for interrupt			
	and exception			
$000\,0777_8$:	vectors	$8^3 - 1 = 511$		
$000\,1000_8$:	„real“ available	$8^3 = 512$	$\longleftarrow$	this is where programs
$\nearrow$	memory			and data start
address in octals				
$\searrow$				stack grows from high
$015\,7777_8$:	$\Uparrow$ stack $\Uparrow$	$56k$ Bytes	$\longleftarrow$	to low locations
$016\,0000_8$:	address	$1 \cdot 8^5 + 6 \cdot 8^4$		
	space for			
$017\,7777_8$:	devices	$1 \cdot 8^5 + (8^5 - 1)$		

2.1.5 Processor Status Word

- The 16-bit processor status word is divided into four parts:

15 through 8	7 through 5	4	3 through 0
...	**Processor Priority**	**Trace Mode**	**Condition Codes**

- Bits 7 – 5 are called the **processor mask** (or interrupt mask), they are used for setting the processor's priority (ranges from 0 to 7):
  - processor priority of 0 to 3 means external interrupts enabled,
  - processor priority of 4 to 7 means external interrupts disabled.

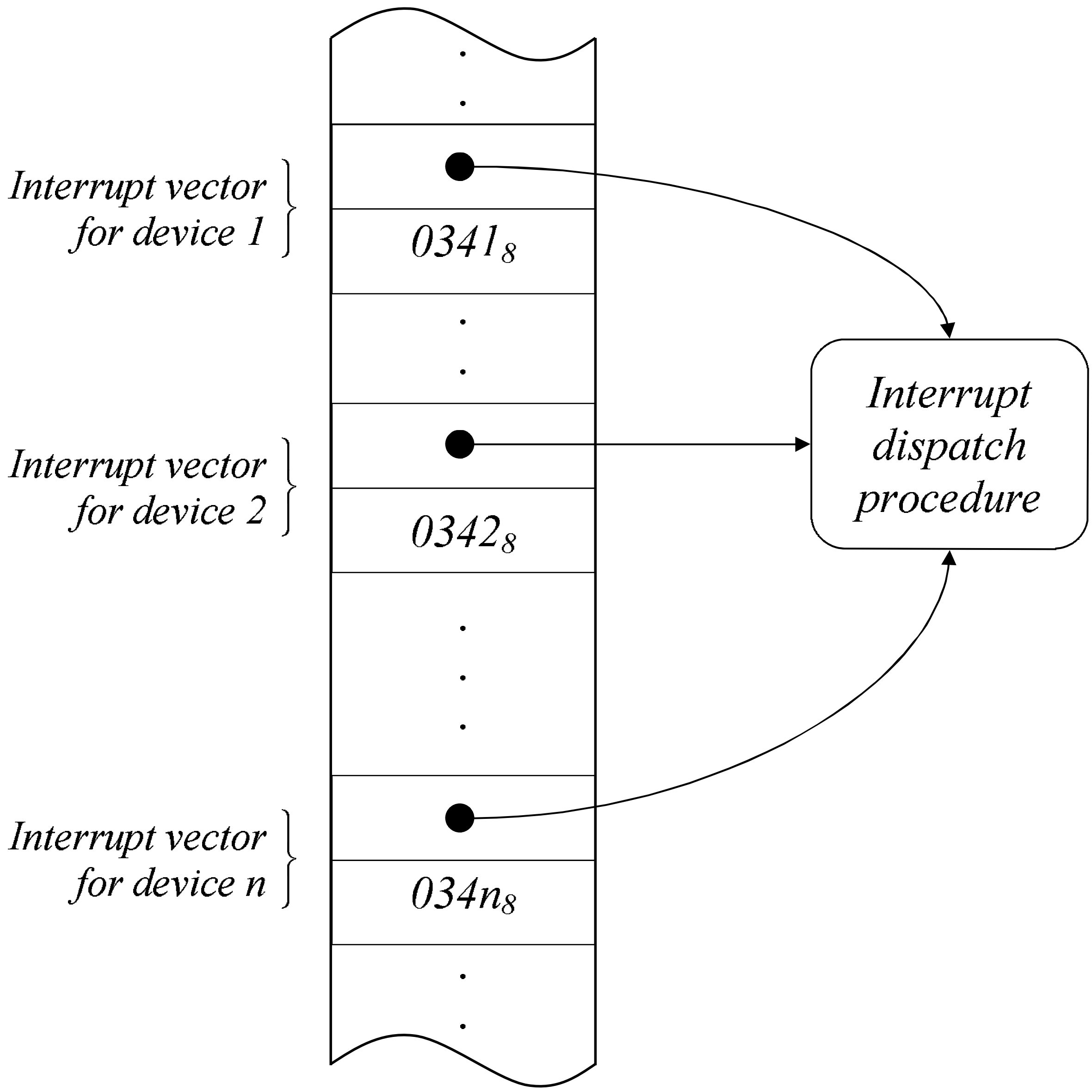
  When needed to disable interrupts, the interrupt level is set to 7 by setting PS's value to $0340_8 = (1 + 2 + 4) \cdot 2^5$.

- Bits 3 – 0 are used for **condition codes**: They are set when e.g. add-instruction produces an arithmetic result, or when comparison is done; records whether result is negative or zero, whether overflow or carry occured; used for conditional bracing instructions.

2.1.6 Vectored interrupts

- The LSI 11/2 uses the **vectored interrupts scheme** for handling exceptions and interrupts from external devices.
- Whenever an external device must communicate with the processor the device **places a signal** on the **interrupt bus line**. If the processor runs with **interrupts enabled**, it checks the interrupt line after executing **each instruction**!
- To handle an interrupt, the processor sends an **acknowledgement** over the bus, requesting the interrupting device to return an **interrupt vector address (IVA)**. The first device with a pending request receives the acknowledgement and responds by returning its interrupt vector address.
- When the CPU receives the address v of the interrupt vector from the Q-bus:
  - the processor pushes the current value of PC and PS on the stack,
  - loads a new PC and PS from two words in memory starting at location v and
  - continues execution at the new location addressed by the PC.

Vectored interrupts



Vectored interrupts

- Each device is assigned a unique IVA, enabling the system software to identify the device, i.e. distinguish between different devices.
- The PC stored at v points to an **interrupt service routine** (interrupt handler) for the interrupting device.
- The interrupt „acts“ like a procedure call **inserted** (invisibily) by the hardware in **between two instructions** in user's code.
- The processor executes the code in the interrupt routine and then returns to the place where the user's process was interrupted.

Vectored interrupts

- To make the interrupt invisible (transparent) to the running program, the interrupt handler must **save and restore the state of the machine**. Saving the state of the 11/2 means saving $R_0 - R_6$, since the hardware saves $R_7 = PC$ and PS.
- In practice the stack pointer R_6 needs **not** to be saved, provided the address of the interrupt routine is **popped off from the stack before returning**, whenever this address has been pushed on the stack before, thus restoring the stack to its original position.
- To prevent an interrupt routine from being interrupted by itself the PS in the IVA specifies a priority level 7, i.e. as soon as **interrupt processing** begins, the processor operates with **interrupts disabled**.

Vectored interrupts

- Interrupt processing ends when the processor executes a **„return from interrupt“** instruction (`rti`, `rtt`), thereby restoring the old values of the PC and PS (before executing the interrupt) are restored from the stack in a **single step**, thereby the SP is restored to its **original value**.
- This amounts to **reversing** the actions taken by the processor when it detected the interrupt, allowing processing to continue where it was interrupted **as if nothing had happened**!

2.1.7 Exceptions

- **Exceptional conditions** are handled like interrupts. Examples for exceptions are: reference to nonexistent memory, execution of invalid instruction code, power failure or other errors detected by the hardware.
- Similarly as above the PC and PS are pushed onto stack and new values for PC and PS are loaded from memory. Exception vectors are **permanently** assigned to memory locations $0 - 27$. Unlike interrupts, they **cannot be disabled** by changing the priority level!

2.1.8 Asynchronous Communication

- An **asynchronous communication device**, the **serial line unit**, sends and receive characters (to/from) a conventional terminal. At least one board in any 11/2 system contains such a device, connecting the computer to a terminal.
- The Cable connecting the device to terminal consists of three wires: one carries data into the computer, one carries data away from the computer and one is the ground voltage wire.
- Signals are **serial** because they travel down the line one bit at-a-time – and they are **asynchronous** because the transmitter sends a character whenever one is available, regardless of the state of the receiver. I.e., there is no synchronization between the transmitter and receiver to control the start of a character.

Asynchronous Communication

- The work to **convert characters** into a **series of pulses** or to covert a series of pulses into characters is carried out by a single integrated circuit (IC) controlled by a precise clock called **Universal Asynchronous Receiver and Transmitter (UART)**.
- **Errors** occur when a receiver can not make **sense** out of the signals it receives. The receiver itself holds **one character at a time**. The CPU must extract **each character** before subsequent characters arrive or the receiver indicates an **overrun condition** and replaces the earlier character with the latest one.

2.1.9 LSI 11/2 Asynchronous Serial Line Hardware

- Conceptually: asynchronous transmitters and receivers are grouped into pairs consisting of one receiver and one transmitter to model two-way communication.

2.1.10 Addressing a Serial Line Unit

- The CPU communicates with a SLU by reading and writing to addresses **beyond the end of real memory**. Boards containing device interfaces watch the bus for addresses corresponding to their devices, and respond accordingly.
- From the programmer's point of view, each SLU contains four 16-bit registers:
  - `RCSR`: receiver control and status registers
  - `RBUF`: received data buffer
  - `XCSR`: transmitter control and status registers
  - `XBUF`: transmitter data buffer
- To communicate with the device, a program **reads or writes the address corresponding to the device register address**, just as it reads or writes to a real memory address.
- Device registers are **not** storage locations! Rather, the interface board containing the device **intercepts** the transfer and **interprets** it as an instruction to control the device or to transfer data between the device and the CPU. So I/O is a side-effect of manipulating certain locations in the address space.

Arrangement of Serial Line Unit Registers

	high-order byte								*low-order byte*							
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RCSR	–	–	–	–	–	–	–	–	**D**	**I**	–	–	–	–	–	–
	(unused)								***receiver control (see below)***		*(unused)*					
RBUF	**E**	**V**	**F**	**P**	**0**	**0**	**0**	**0**	*	*	*	*	*	*	*	*
	receiver errors								***character***							
XCSR	–	–	–	–	–	–	–	–	**T**	**I**	–	–	–	–	–	**B**
	(unused)								***transmitter control (see below)***		*(unused)*					
XBUF	–	–	–	–	–	–	–	–	*	*	*	*	*	*	*	*
	(unused)								***character***							

`RCSR`	Bit 7 (**D**)	1 ≙ character received from SLU
`RCSR`	Bit 8 (**I**)	1 ≙ SLU posts interrupt when character received
`XCSR`	Bit 7 (**T**)	1 ≙ free to transmit, character has been sent
`XCSR`	Bit 8 (**I**)	1 ≙ SLU has received character (finished sending) and new character can be written

Examples of addressing a Serial Line Unit

- **Writing** a character to the `XBUF` register causes the SLU to capture that character and **transmitting** it down to the SLU. Once the board accepts a character, it operates **independently** of the CPU (processing continues while the transmitter sends the character).
- The CPU receives a character received over the SLU by **reading from the `RBUF` address**:

15	14	13	12	11 10 9 8	7 6 5 4 3 2 1 0
error occured	overflow error	framing error	parity error	(unused)	**character**

- `RBUF` is **read only** – write instructions directed to that address are ignored. Accessing RBUF has a **side effect**: the receiver is cleared an enabled to receive the next character.

2.1.11 Polled vs. Interrupt-Driven I/O

$\longrightarrow$ discussed later in chapter 13

2.2 Disk Storage Organization

$\longrightarrow$ discussed later in chapter 17

2.3 The C Run-Time Environment

- Operating Systems are written in **high-level language** to make them **easier** to write, understand, debug, and move to other machines.
- However, sometimes **machine assembly language** procedures are introduced because **machine quantities** must be manipulated, e.g., for saving and restoring the machine's registers by context-switching code, interrupt routines, manipulating the process table, implementing semaphores, operating on the real-time clock, and so on.

The C Run-Time Environment

- The C-Compiler expects each program to run in an address space as the following:

↓ 0	↓_etext	↓_edata	↓_end
text	**data**	**bss**	**free space** ⟵ ··· **stack**

  - **text** segment contains text of main program and all its procedures
  - **data** segment contains all initialized data
  - **bss** segment contains all uninitialized data
  - the **stack** occupies the highest part of address space and grows downwards
  - **heap storage** ist allocated from `_end` upwards, i.e., it grows from the bottom of the free space upwards.
- Symbols `_etext`, `_edata`, `_end` refer to global variables inserted into object code by the loader. They are initialized to the first address beyond **text**, **data** and **bss** segments, respectively.
- The running program can find out how much free memory remains between the end of loaded part and the current top of the stack by taking the address of `_end`.

The C Run-Time Environment

- When Xinu runs multiple processes, it allocates a **separate stack** from the **highest available free space** for each process:

↓ 0	↓_etext	↓_edata	↓_end			
text	**data**	**bss**	**free**	**stack** # 3	**stack** # 2	**stack** # 1

- **Stack overflow** in one process destroys stacks of other processes – techniques to prevent this from happening are discussed in chapter 15.
- Stack space is **returned** to the free list whenever a process exits.

Conventions for translating procedures in the C-compiler

- Consider a **calling procedure** A and a **called procedure** B:

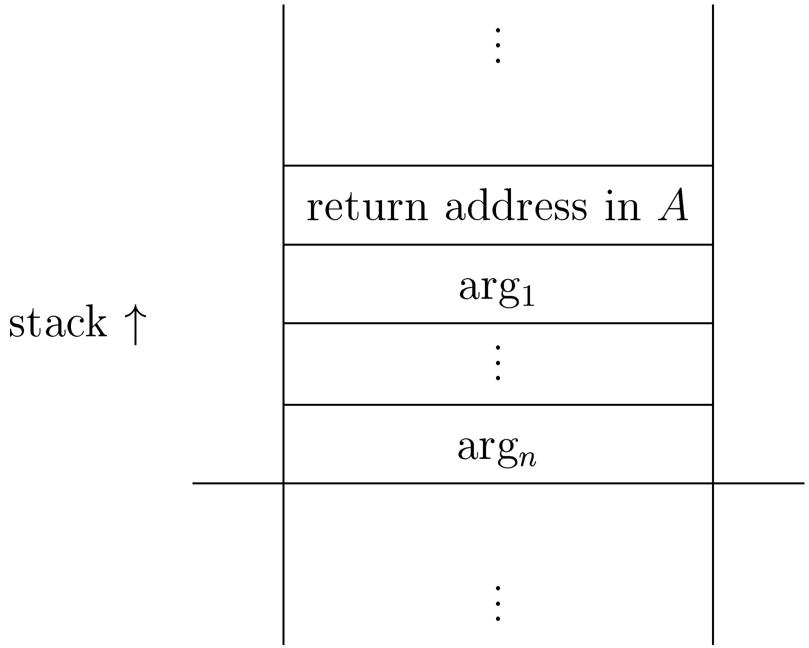
```
proc A:
    ...; B(arg_1, ..., arg_n); ...
```

- How is the code of B within the body of A implemented in C's compiler?
  1. Values of actual **parameters** of B are pushed on stack (in **reverse** order).
  2. **Return address** in A, i.e., of the instruction following the call to B, is pushed on the stack.
  3. **Flow control** branches to B.

  The latter two are carried out by calling a **single instruction**: `jsr pc,` $address_B$.

- Calling procedure A is also responsible for **popping** B**'s arguments** from the stack after the called procedure B returns!

Image of Stack



Conventions for the called procedure B

- The called procedure B is **responsible** for
  - **saving and restoring the machine state**, i.e., the contents of the registers
  - **popping the return address** from the stack before returning
- Saving the contents of the used registers is done by calling the assembly language routine `csv` (**C-registers save routine**).
- Restoring the contents of the used registers is done by calling the assembly language routine `cret` (**C-registers restore routine**).
- Hardware provides a **single instruction `rts pc`** that pops an address from the stack and returns to that address.

Schematic translation of calling procedure A

(1) ... code for A ...

(2) code for putting the arguments of B's call on the stack in reverse order

(3) `jsr pc,` $address_B$

— this is where B is executed —
(original value of PC points to 4.)

→(4) code for popping B's arguments from the stack

(5) ... code for A ...

(6) `rts pc` (to return to caller of A)

Schematic translation of called procedure B

(1) `move r5, r0` (code for pushing the old value of R_5 on the stack)

(2) `move pc+2, r5` (assigning to R_5 the return address in B after calling `csv`)

(3) `jsr r0, csv` (jumping to `csv`)

(4) ... code for B ...

— Assumption: after executing fragment (4) of B's code,
the stack has the same contents, and the values of SP
and R_5 are the same es before executing fragment (4) —

(5) `rts pc` (pops address of `cret` from the stack and returns to `cret`)

Image of Stack

	$\vdots$	
	address of `cret`	done by `csv`
	old value of R_2	done by `csv`
	old value of R_3	done by `csv`
	old value of R_4	done by `csv`
stack $\uparrow$	old value of R_5	`jsr r0, csv`
	return address in A	`jsr pc,` *address*$_B$
	arg_1	
	$\vdots$	
	arg_n	
	$\vdots$	

C-registers save routine `csv` and C-registers restore routine `cret`

- To save the machine registers, the compiler inserts a call to `csv`, an assembly language routine. `csv`
  - **saves the registers** and
  - **jumps back** to called routine B.
- When code for B is finished, compiler inserts a call to assembly language routine `cret` to restore the values of the machine's registers and the stackpointer SP ($= R_6$) and to return control to the original caller A. `cret`
  - restores **old values of machine registers**
  - restores the **stack pointer**
  - **returns control** to original caller

C-registers save routine `csv` and C-registers restore routine `cret`

`csv:`

(1) $R_0 := R_5$ (by assumption R_5 contains return address in B, and stack has old value of R_5, the return address in A, and B's arguments on it)

(2) $R_5 := SP$ (R_5 stores value of SP, i.e., address of B's frame)

(3) R_4, R_3, R_2 pushed on stack in that order

(4) `jsr pc, (r0)` (old value of PC, i.e., address of `cret`, pushed on stack and control branches to address stored in R_0, i.e., return address in B)

`cret:`

(1) $R_2 := R_5$ (put copy of called frame pointer in R_2, see (2) above in `csv`)

(2) restore R_4, R_3, R_2 in that order

(3) $SP := R_5$ (restore stack pointer)

(4) `mov (sp)+, r5` (restore R_5 on stack by call to `csv` procedure entry)

(5) `rts pc` (return to A)

C-registers save routine `csv` and C-registers restore routine `cret`

```
/* csv.s - csv, cret */

/* C register save: upon entry here, procedure A has called B   */
/* and B has called csv to save registers. r5 contains return   */
/* address in B. The stack has old r5, return address in A, and */
/* arguments on it. C return: cret (below) is used to restore   */
/* regs when the called proc. exits                             */

      .glob csv, cret
/*------------------------------------------------------------------*/
/* csv  -- C register save routine
/*------------------------------------------------------------------*/
csv:
      mov   r5,r0     / r0 not saved at call (C convention)
      mov   sp,r5     / r5 points to called routine's frame
      mov   r4,-(sp)  / push r4 - r2 on stack
      mov   r3,-(sp)
      mov   r2,-(sp)  / jsr pushes PC onto stack hoes to
      jsr   pc,(r0)   / address in r0 (originally in r5)

/*------------------------------------------------------------------*/
/* cret  -- C register restore routine
/*------------------------------------------------------------------*/
cret:
      mov   r5,r2     / put copy of called frame ptr in r2
      mov   -(r2),r4  / reload r4 - r2 from start of frame
      mov   -(r2),r3
      mov   -(r2),r2
      mov   r5,sp     / restore SP
      mov   (sp)+,r5  / restore r5 on stack by call
                      / to csv procedure entry
      rts   pc        / return to caller
```