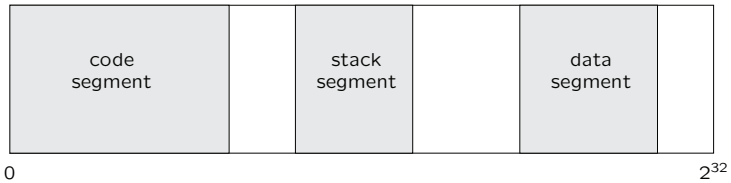
Virtual memory

Managing the memory used by a process

- A process needs memory for
  - The executable code,
  - The stack,
  - If needed, other large amounts of data.
- The amount of memory needed by a process is not always known *a priori* and can vary while the process is executed.
- To reserve a fixed amount of memory for a process is too restrictive from the point of view of the process (constraint on the usable memory addresses) and from the point of view of the system (lack of flexibility with respect to resource management).

A virtual view of memory

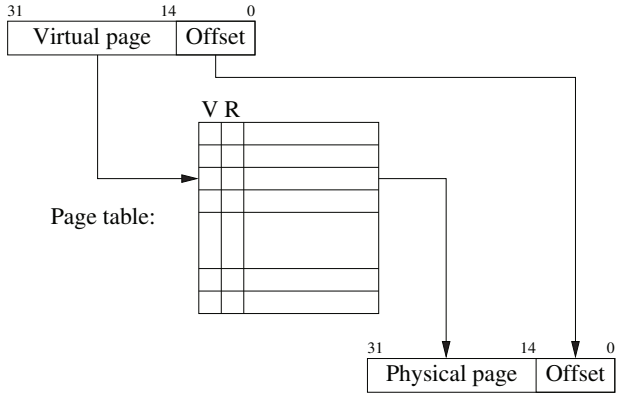
- Each process sees its own virtual memory space that has the size of the addressable space for the machine being used (2^{32} bytes for β).
- In this virtual memory space the process asks for the allocation of memory *segments* (contiguous spaces) that can then be used.
- The memory addresses within an allocated segment are translated to physical addresses; accessing other addresses is considered to be an error.



Implementing virtual memory: paging

- To implement virtual memory, the address space is divided into *pages*, for example 2^{18} pages of 4K words for the ULg version of the β machine.
- Only a selection of pages of the virtual space of a process are in physical memory.
- The correspondence between virtual and physical pages is embodied in a *page table*.
- To extend available memory, some pages can be saved to disk.

Translating from virtual to physical addresses



Besides the physical address, the page table includes on the validity of pages. The bit `V(alid)` indicates whether the page has been allocated or not, the bit `R(esident)` whether the page is resident in memory or saved on disk.

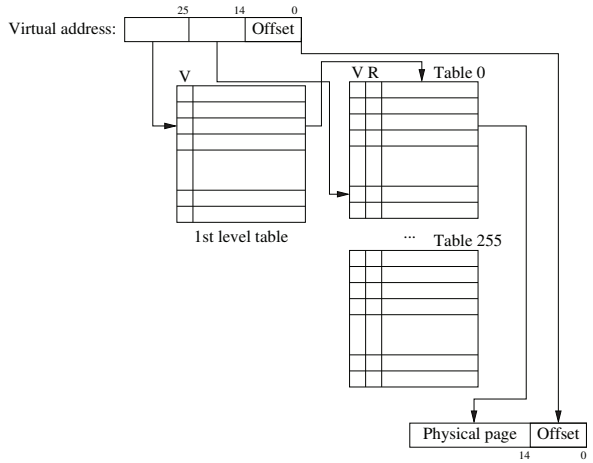
The page table: the speed problem

- For each memory access, one needs to use the page table, which itself is also . . . in memory.
- Thus accessing virtual memory requires accessing virtual memory. A simple way to avoid this circularity is to keep the page table in kernel memory, which is then managed directly as physical memory.
- Even if circularity is avoided, multiplying memory accesses substantially slows a machine down.
- To avoid this, one keeps part of the page table in fast memory. This is called a page table *cache* or a *Translation Lookaside Buffer* (TLB).
- If the address of the accessed page is not in the cache, one says that there is a *cache miss*

The page table: the size problem

- A page table for an addressable space of 2^{32} bytes and 4K word pages includes 2^{18} entries and a size of the order of 1 MBytes. However, only a small of this table will be actually used.
- For the page table, one can thus use an appropriate data structure (hash table, tree) that limits the required space while preserving access speed.
- Another commonly used solution is to have a two-level table.

A two-level page table



Only the actually used second level tables are allocated.

Paging: why does it work?

- If memory accesses were completely random, a page table cache would not be very useful and virtual memory would not be usable.
- What allows virtual memory to operate effectively is that memory accesses satisfy a locality principle: accesses that are time-wise close are often also address-wise close.
- This is clearly true for code addresses, but is also often true for data addresses.

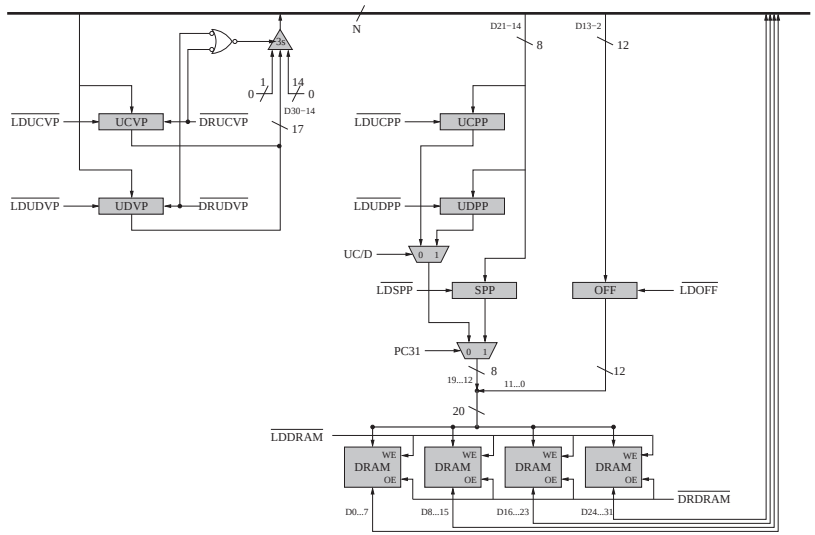
Paging and processes

- To separate the memory used by different processes sharing a machine, an elegant solution is for each process to have its own page table.
- To switch from one process to another, it is then sufficient to restore registers and to switch page table.
- Since the system manages the page tables and the allocation of pages to processes, it is impossible for a process to have access to another process's memory. Processes thus work in separate isolated *contexts*.
- It is possible to have several page table caches and thus to have several contexts for which fast access is possible.

An implementation of virtual memory in the β architecture: ULg03

- User processes work in virtual memory, each in its own context; the kernel works directly in physical memory.
- There are two contexts available in hardware: a user context and a kernel context (physical memory).
- The page table cache is divided into two parts, each containing just a single page address: one part of data memory accesses (`LD` and `ST`); one part for accessing instructions (next instruction, `JMP` and `BR`).
- There are special instructions, only accessible in supervisor mode, for accessing the page table cache.
- The page tables are kept in kernel memory. Accessing a page whose address is not in the cache triggers an exception that is handled by the kernel.

The dynamic RAM module of ULg03



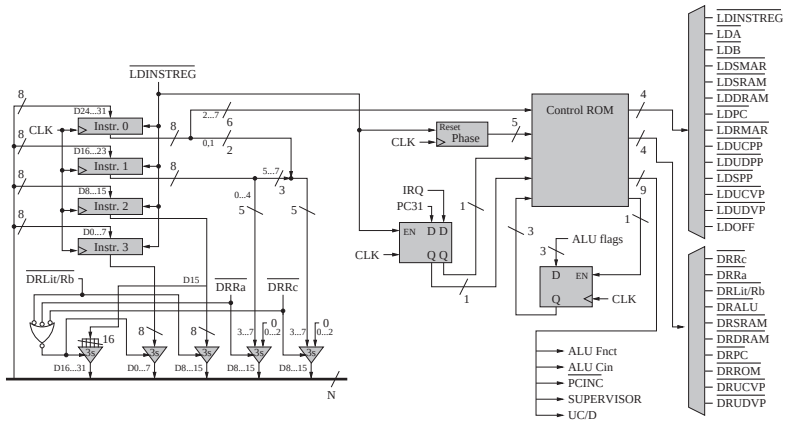
The dynamic RAM module of ULg03 (next)

- The register names are the following :
  `UCVP`, `UCPP` : User Code Virtual/Physical Page.
  `UDVP`, `UDPP` : User Data Virtual/Physical Page.
  `SPP` : System Physical Page

- In supervisor mode (`PC31` at 1), the addresses used are physical addresses.

- In user mode (`PC31` at 0), depending on whether one is accessing a code or a data page (signal `UC/D`), one uses the page address `UCPP` or `UDPP`, provided that the virtual page number coincides with the content of `UCVP` (`UDVP`), which is checked by the microcode. Otherwise, an exception is triggered.

The control unit of ULg03



- Up to 32 phases are now allowed (longer microcode).
- New `LD` and `DR` signals are introduced to control the DRAM address registers; new signal `UC/D`.

The constant ROM of ULg03

0x00	⋮
	⋮
0xFA	10000000 00000000 00100000 00001100 *(Address of the handler "IRQ" — 0x2012)*
0xFB	10000000 00000000 00100000 00001000 *(Address of the handler "Supervisor Call" — 0x2008)*
0xFC	10000000 00000000 00100000 00010000 *(Address of the handler "Illegal Operation" — 0x2016)*
0xFD	10000000 00000000 00100000 00000100 *(Address of the handler "Cache Miss Code" — 0x2004)*
0xFE	10000000 00000000 00100000 00000000 *(Address of the handler "Cache Miss Data" — 0x2000)*
0xFF	00000000 00000000 11110000 00000000 *(Address of register XP)*

The constant ROM of ULg03 (next)

- There are two new addresses: *Cache Miss Data* and *Cache Miss Code*.

- The handler addresses are consecutive (the handlers start with a "branch").

The microcode of ULg03

- Each instruction is now different in supervisor and user modes (there always is a memory access to load the next instruction).
- In supervisor mode, the addresses are physical and can thus be used directly.
- In user mode, the page address is compared to the one found in `UC(D)VP`. If it coincides, the physical page found in `UC(D)PP` is used. If not an exception is triggered.
- On a code *cache miss*, one returns to the next instruction (or to the jump target); on a data *cache miss*, the instruction is reexecuted.

  The `JMP` instruction is an interesting case since it allows to move from a physical address (user mode) to a virtual address (user mode).

The microcode of `LD`

`LD(Ra, Literal, Rc)` (user mode)

Opcode = 011000 IRQ = 0 PC31 = 0

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU $F,\overline{C_{in}},M$	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0001	0	0	SMAR <- Ra
00001	*	1	0	000000	0010	0100	0	0	B <- SRAM
00010	*	1	0	000000	0001	0010	0	0	A <- Lit
00011	*	1	0	100110	0001	0011	0	0	A <- A+B
00100	*	1	0	000000	0010	1001	0	0	B <- UDVP
00101	*	1	0	111111	1101	0011	0	0	OFF <- A
00110	*	1	0	111111	1100	0011	0	0	UDVP <- A
00111	*	1	0	000000	0001	1001	0	0	A <- UDVP
01000	*	0	0	100101	0010	0011	0	0	B <- A /XOR B ; Latch

`A` contains only 1's iff the virtual page number coincides with the one held in the cache.

There is a data *cache miss*; one must jump to the corresponding handler and return to `XP -4`, which means that the instruction will be reexecuted.

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU $F,\overline{C_{in}},M$	LD SEL	DR SEL	PC+	SVR	
01001	E=0	1	0	110011	0001	0011	0	0	A <- 0xFFFFFFFF
01010	E=0	1	0	111111	0111	0011	0	0	RMAR <- A
01011	E=0	1	0	000000	0011	0111	0	0	SMAR <- ROM
01100	E=0	1	0	000000	0100	0110	0	0	SRAM <- PC
01101	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
01110	E=0	1	0	000000	0110	0111	0	1	PC <- ROM; SVR
01111	E=0	1	0	000000	1010	0110	0	0	SPP <- PC
10000	E=0	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
10001	E=0	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

Note that from phase `01110`, `PC31` has value 1 and thus physical addresses are used.

There is no data *cache miss*; the next instruction has to be loaded, checking that there is no code *cache miss*.

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU $F,\overline{C_{in}},M$	LD SEL	DR SEL	PC+	SVR	
01001	E=1	1	0	000000	0011	0000	0	0	SMAR <- Rc
01010	E=1	1	1	000000	0100	0101	0	0	SRAM <- DRAM; UC/D
01011	E=1	1	0	000000	0010	1000	0	0	B <- UCVP
01100	E=1	1	0	000000	0001	0110	0	0	A <- PC
01101	E=1	1	0	111111	1011	0011	0	0	UCVP <- A
01110	E=1	1	0	000000	0001	1000	0	0	A <- UCVP
01111	E=1	1	0	111111	0001	0011	0	0	A <- A
10000	E=1	1	0	111111	0001	0011	0	0	A <- A
10001	E=1	0	0	100101	0010	0011	0	0	B <- A /XOR B ; Latch

There is a code *cache miss*, one must jump to the corresponding handler and return to XP, which means that the following instruction will be executed.

Phase	Fl.	Latch flags	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
10010	E=0	1	0	110011	0001	0011	0	0	A <- 0xFFFFFFFF
10011	E=0	1	0	111111	0111	0011	0	0	RMAR <- A
10100	E=0	1	0	000000	0011	0111	0	0	SMAR <- ROM
10101	E=0	1	0	000000	0100	0110	0	0	SRAM <- PC
10110	E=0	1	0	111110	0001	0011	0	0	A <- A-1
10111	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
11000	E=0	1	0	000000	0110	0111	0	1	PC <- ROM; SVR
11001	E=0	1	0	000000	1010	0110	0	0	SPP <- PC
11010	E=0	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
11011	E=0	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

Note that from phase 26, PC31 has value 1.

There is no code *cache miss*; one moves to the next instruction.

Phase	Fl.	Latch flags	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
10010	E=1	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
10011	E=1	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

The microcode of JMP

JMP(Ra, Rc) (user mode)

Opcode = 011011 IRQ = 0 PC31 = 0

Phase	Fl.	Latch flags	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0001	0	0	SMAR <- Ra
00001	*	1	0	000000	0001	0100	0	0	A <- SRAM
00010	*	1	0	000000	0011	0000	0	0	SMAR <- Rc
00011	*	1	0	000000	0100	0110	0	0	SRAM <- PC
00100	*	1	0	000000	0010	1000	0	0	B <- UCVP
00101	*	1	0	111111	1011	0010	0	0	UCVP <- A
00110	*	1	0	111111	1101	0010	0	0	OFF <- A
00111	*	1	0	111111	0110	0001	0	0	PC <- A
01000	*	1	0	000000	0001	1000	0	0	A <- UCVP
01001	*	0	0	100101	0010	0011	0	0	B <- A /XOR B; Latch

No code *cache miss*, load the JMP target

Phase	Fl.	Latch flags	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
01010	E=1	1	0	000000	0000	0101	1	0	INSTREG <- DRAM; PC+

Code *cache miss*, jump to the corresponding handler.

Phase	Fl.	Latch flags	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
01010	E=0	1	0	110011	0001	0011	0	0	A <- 0xFFFFFFFF
01011	E=0	1	0	111111	0111	0011	0	0	RMAR <- A
01100	E=0	1	0	000000	0011	0111	0	0	SMAR <- ROM
01101	E=0	1	0	000000	0100	0110	0	0	SRAM <- PC
01110	E=0	1	0	111110	0001	0011	0	0	A <- A-1
01110	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
10000	E=0	1	0	000000	0110	0111	0	1	PC <- ROM; SVR
10001	E=0	1	0	000000	1010	0110	0	0	SPP <- PC
10010	E=0	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
10011	E=0	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

JMP(Ra, Rc) (supervisor mode)

Opcode = 011011 IRQ = * PC31 = 1

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0001	0	0	SMAR <- Ra
00001	*	1	0	000000	0001	0100	0	0	A <- SRAM
00010	*	1	0	000000	0011	0000	0	0	SMAR <- Rc
00011	*	1	0	000000	0100	0110	0	0	SRAM <- PC
00100	*	0	0	111111	0011	0010	0	0	A <- A; Latch

Jump to an address whose most significant bit is 1 : one stays in supervisor mode and handles a physical address.

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00101	N=1	1	0	111111	0110	0010	0	1	PC <- A; SVR
00110	N=1	1	0	000000	1010	0110	0	0	SPP <- PC
00111	N=1	1	0	000000	1101	0110	0	0	OFF <- PC; PC+
01000	N=1	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

Jump to an address whose most significant bit is 0 : one switches to user mode and handles a virtual address that has to be translated if needed. From phase 9, PC31 has value 0.

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00101	N=0	1	0	000000	0010	1000	0	0	B <- UCVP
00110	N=0	1	0	111111	1011	0010	0	0	UCVP <- A
00111	N=0	1	0	111111	1101	0010	0	0	OFF <- A
01000	N=0	1	0	111111	0110	0001	0	0	PC <- A
01001	N=0	1	0	000000	0001	1000	0	0	A <- UCVP
01010	N=0	0	0	100101	0010	0011	0	0	B <- A /XOR B ; Latch

From there on the microcode is the same as for a user mode JMP.

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
01011	E=1	1	0	000000	0000	0101	1	0	INSTREG <- DRAM; PC+
01011	E=0	1	0	110011	0001	0011	0	0	A <- 0xFFFFFFFF
01100	E=0	1	0	111111	0111	0011	0	0	RMAR <- A
01101	E=0	1	0	000000	0011	0111	0	0	SMAR <- ROM
01110	E=0	1	0	000000	0100	0110	0	0	SRAM <- PC
01111	E=0	1	0	111110	0001	0011	0	0	A <- A-1
10000	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
10001	E=0	1	0	000000	0110	0111	0	1	PC <- ROM; SVR
10010	E=0	1	0	000000	1010	0110	0	0	SPP <- PC
10011	E=0	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
10100	E=0	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

The supervisor mode special instructions

Four new instructions, only available in supervisor mode, are introduced in order to make the implementation of the *cache miss* handlers possible.

Opcode	nom	définition
001000	RDUCVP(Rc)	Rc <- UCVP
001001	RDUDVP(Rc)	Rc <- UDVP
001010	WRUCPP(Rc)	UCPP <- Reg[Rc]
001011	WRUDPP(Rc)	UDPP <- Reg[Rc]
001100	WRUCVP(Rc)	UCVP <- Reg[Rc]
001101	WRUDVP(Rc)	UDVP <- Reg[Rc]

The corresponding macros are.

```
.macro RDUCVP(Rc)          ENC_NOLIT(0b001000,0,0,Rc)
.macro RDUDVP(Rc)          ENC_NOLIT(0b001001,0,0,Rc)
.macro WRUCPP(Rc)          ENC_NOLIT(0b001010,0,0,Rc)
.macro WRUCPP(Rc)          ENC_NOLIT(0b001011,0,0,Rc)
.macro WRUCVP(Rc)          ENC_NOLIT(0b001100,0,0,Rc)
.macro WRUCVP(Rc)          ENC_NOLIT(0b001101,0,0,Rc)
```

The microcode of RDUCVP

RDUCVP(Rc) (suepervisor mode) : Rc ¡- UCVP

Opcode = 001000 IRQ = * PC31 = 1

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0000	0	0	SMAR <- Rc
00001	*	1	0	000000	0100	1000	0	0	SRAM <- UCVP
00010	*	1	0	000000	1010	0110	0	0	SPP <- PC
00011	*	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
00100	*	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

We are in supervisor mode and thus working with physical addresses.

A handler for the data "cache miss" exception

The handler starts with the following stub.

```
h_stub:  SUBC(XP, 4, XP)       | plan to reexecute the instruction
                               | that has been suspended
         ST(r0, User, r31)     | save
         ST(r1, User+4, r31)
              . . .
         ST(r30, User+30*4)
         CMOVE(KStack, SP)     | Load the system SP
         RDUDVP(r1)            | the virtual page address to be
                               | translated
         PUSH(r1)              | pass it as argument

         BR(CMHandler,LP)      | call the  Handler
         DEALLOCATE(1)         | remove the argument from the stack
         WRUDPP(r0)            | install the returned value
         LD(r31, User, r0)     | restore
         LD(r31, User+4, r1)
         LD(r31, User+30*4, r30)
         JMP(XP)               | return to application
```


The C part of the handler is the following.

```
struct PTentry {short valid, resid ; int PhysPage} PageMap[262144]:
      /* The page table of the current process, 2^18 entries */

CMHandler(VpageNo)
{ if (PageMap[VpageNo].valid != 0 && PageMap[VpageNo].resid != 0)
       return PageMap[VpageNo].PhysPage ;
     else Pagerror(VpageNo);
}
```

The error can be due to accessing a page that has not been allocated (segmentation fault), or to a page saved to disk that has to be restored to memory (page fault).