RAL-PPD Academic Training 2007-2008

Computer – Hardware Interactions
(with emphasis on VME)

Norman Gee

5-Mar-2008

Science & Technology Facilities Council

Overview

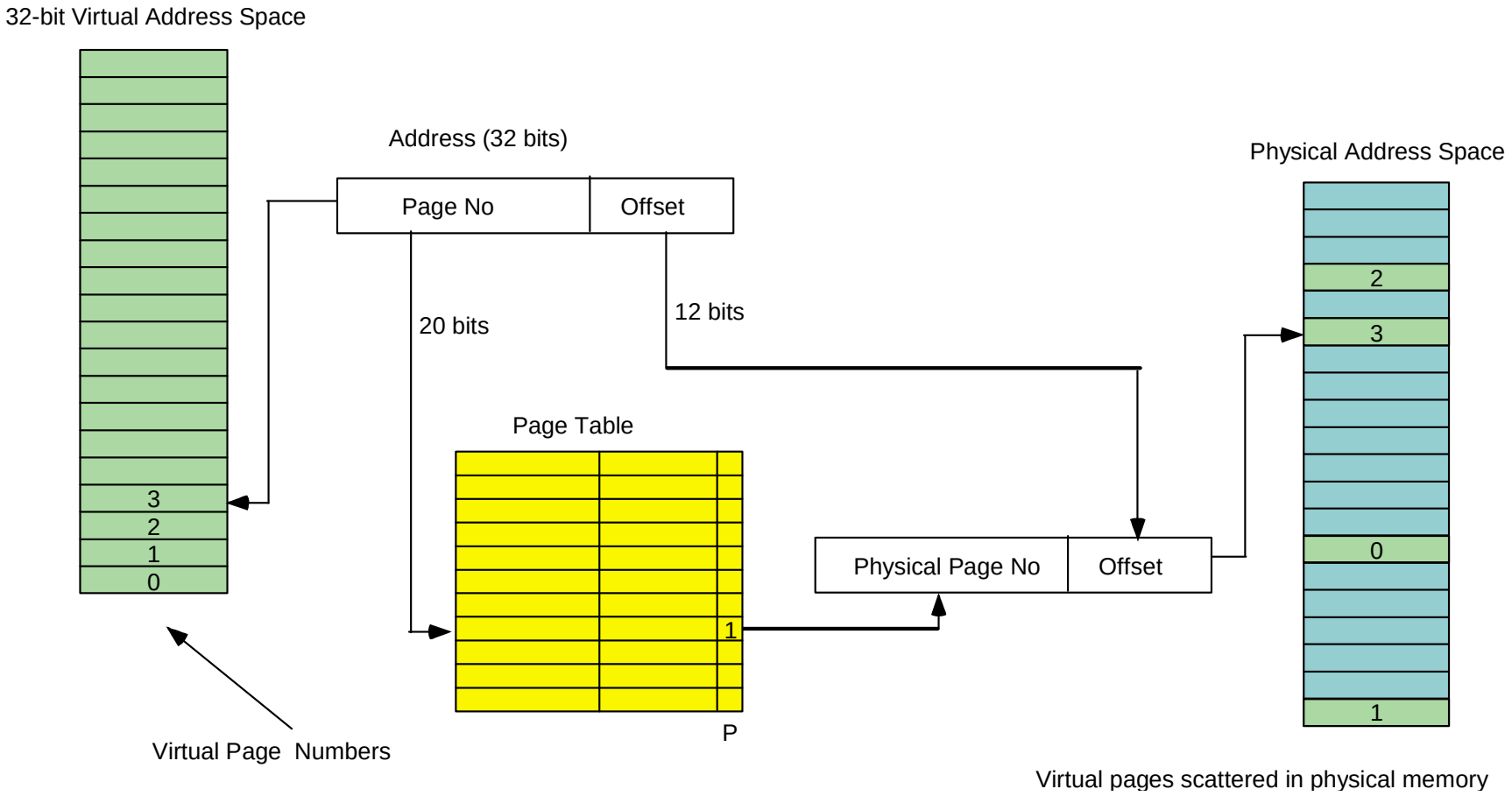
- Classes of Interaction
- Mapped I/O
- Hardware - VMEbus, with a short interlude on VME cycle internals
  - *Interfacing between Computer and VMEbus*
  - *DMA vs Block Transfer, VME extensions*
- Hints and Tips
- Summary
- Bibliography
- Q&A

Classes of Interaction

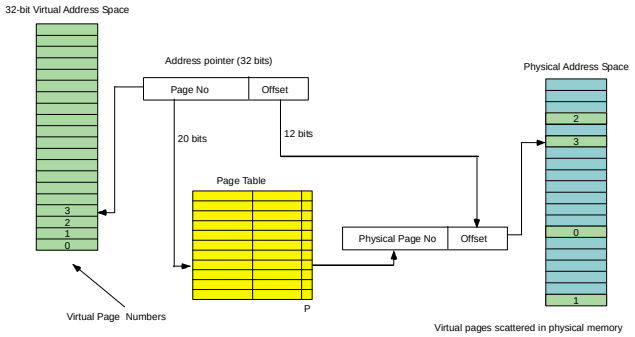
	CPU – Controller	CPU – Slave (memory)
Distances	At least several metres	0.3m or less
Media	Cable, Fibre;	Traces on PC boards & backplanes;
Format	Serial data	Parallel data
Speed and common examples:		
Slow (to 1Mb/s)	(RS232), CANbus, GPIB	(CAMAC)
Medium (to 100 MB/s)	USB, Firewire, Ethernet	VME, PCI
Fast (100 MB/s +)	Gigabit Ethernet, Telecoms	PCI-X, PCI-e

Licence to crash system

Mapped I/O - Virtual Memory

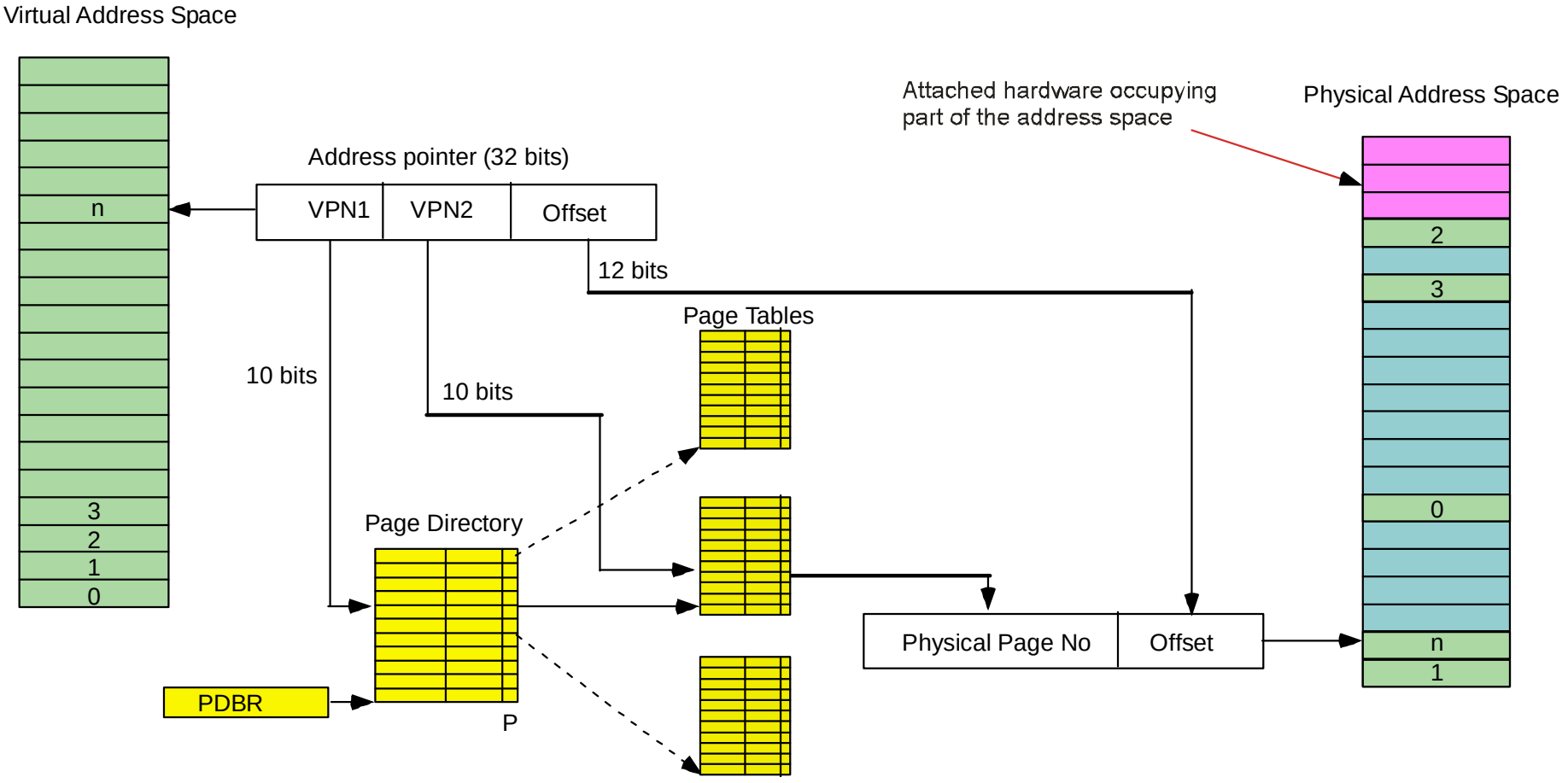


Virtual Memory

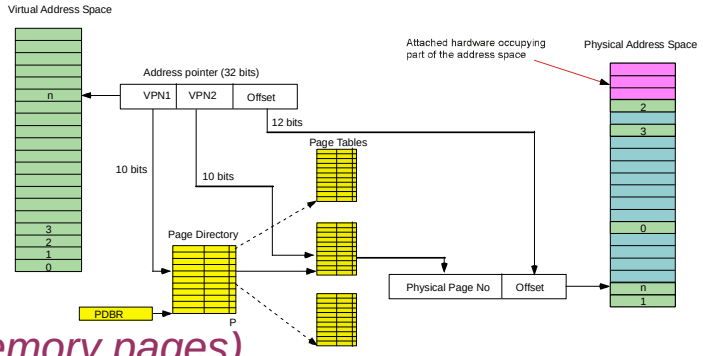


- Note:
  - *Some virtual memory isn't in physical memory (paging, swapping)*
  - *Some physical memory isn't visible from your code (unmapped or protected)*
  - *Virtually contiguous pages are normally physically discontiguous*
  - *The page table itself is held in memory, so virtual-to-physical translation involves an extra memory access… which takes time*
    - *The CPU has a cache of translated (page) addresses - "Translation Lookaside Buffer" TLB*
    - *A program needs several translated addresses at once (program code, data areas, stack,…)*
  - *The stack normally lives in high virtual memory so that it can expand downwards*
    - *The page table has to cover the whole 32-bit virtual address range, so gets BIG*

Segmented Page Tables and Hardware

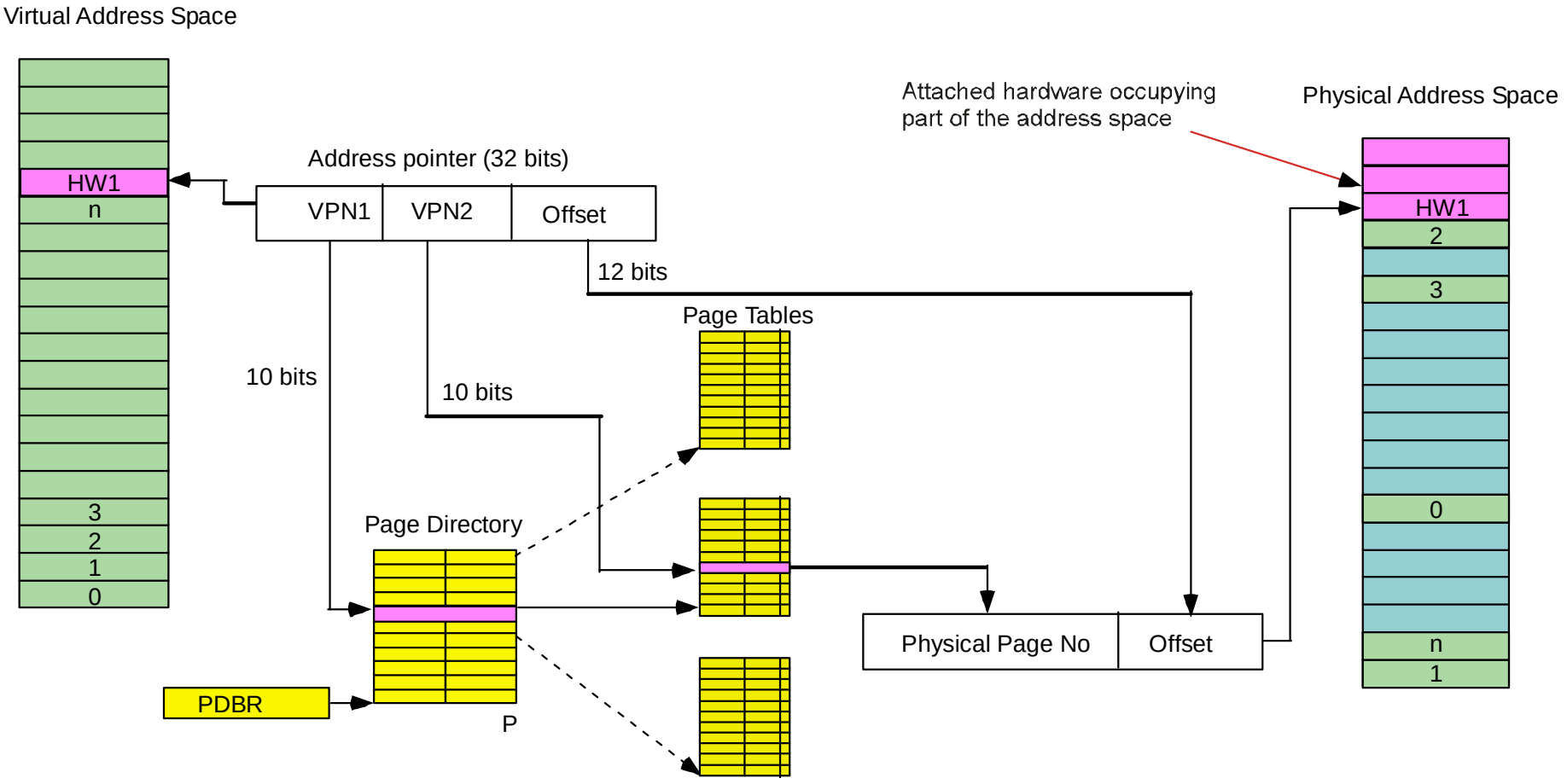


Segmented Page Tables and Hardware



- Note:
  - *Hardware appears in the physical address space, looking rather like memory*
  - *Placed at physical addresses above memory*
  - *Consumes complete pages (no mixed hardware/memory pages)*
  - *The physical hardware addresses may be fixed in the hardware, or may be controlled by registers or maps in the hardware*
    - *…which the operating system must set up*
  - *To be visible to your program, your page tables must point to the hardware addresses*
    - *Your program (or the operating system on your behalf) will see the hardware at virtual addresses.*
      - *Other programs may see the same hardware (via <u>their</u> page tables) at different virtual addresses*
    - *Most user programs are not mapped to most hardware, or the pages are protected*

Accessing Mapped Hardware

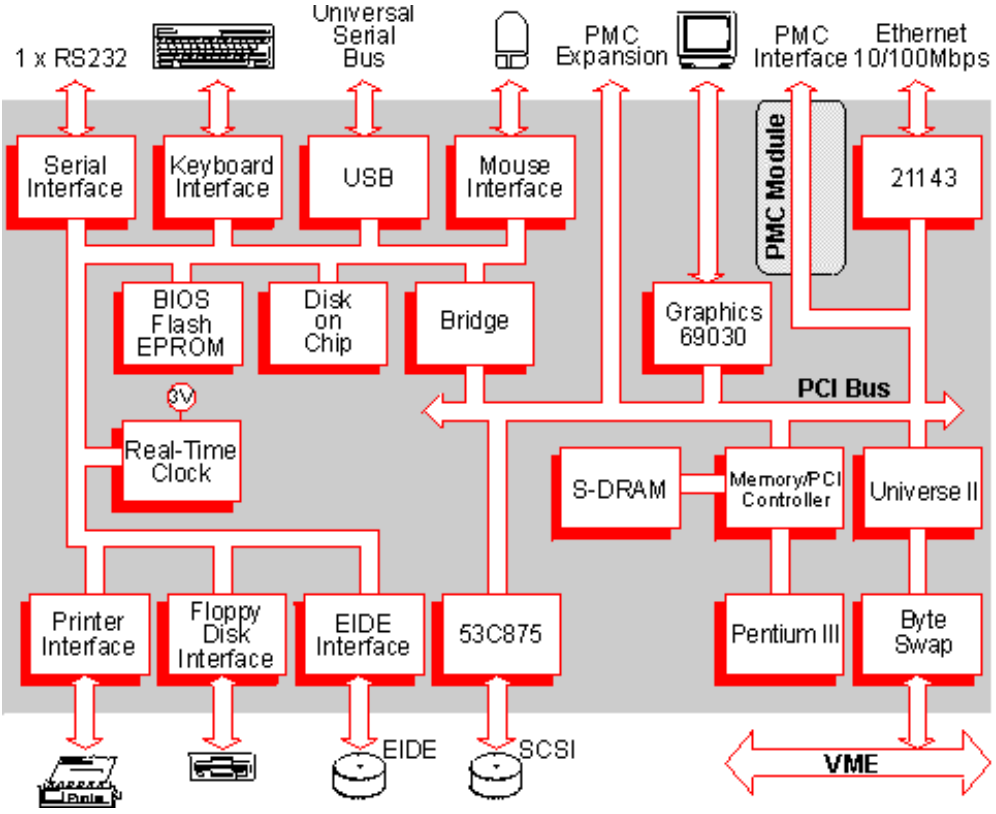


Real Hardware

- Can connect hardware in a variety of different ways…
  - *Specialised: PCI bus interfaces to e.g. SCSI, IDE (disks);*
  - *General purpose: PCI bus interfaces to other formats - e.g. USB, Firewire,…*
- The interface will have internal addresses and may map external ones (this is where controller and slave media diverge)
- Hardware needing only few addresses will be allocated a small fixed range of physical addresses (hardwired or at boot time)
- For a larger address range, hardware may have a second layer of mapping logic (like memory) to determine where device addresses appear in system address space
- If the Hardware has a really big address space, there may not be enough spare physical address space to map all of it at the same time
  - *…so you may have to change the mapping for every access*

The interface from Computer … to VME

- Most VME single board computers use the Tundra “Universe” chip to link to VME
- Connects to the CPU via PCI bus
  - *R/W cycles (address, data)*
- Two connections/maps involved:
  - *CPU virtual to physical*
  - *PCI (=physical) to VME*
- CPU read/write instructions translate directly to PCI cycles and thence to VME cycles



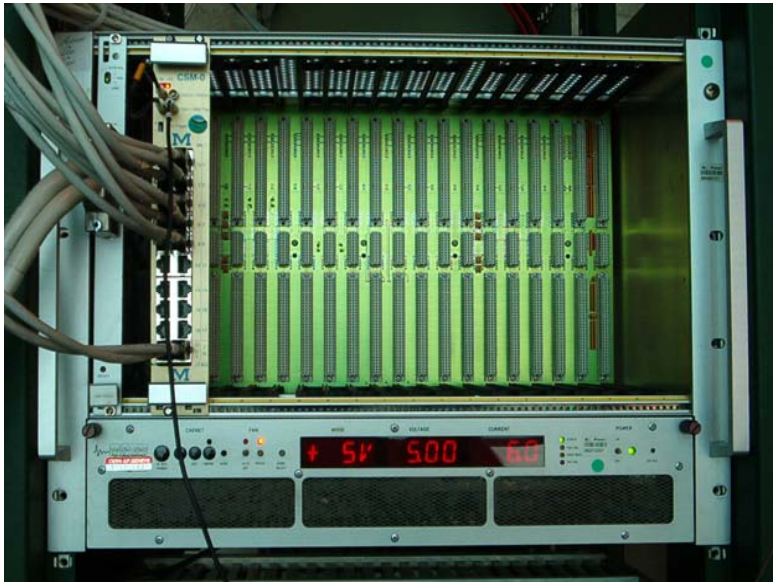
Summary so far…

- Most common computer connections expect an isolated controller at the far end of a link
  - *but VME is not like this, the slave modules can be densely packed and very stupid*
- Hardware has to appear in system address space
  - *… looking like memory*
- The computer's memory management tables must map the hardware into virtual address space
  - *… so that your software can see it*

- The next section exposes some details of VME, its protocols and its mapping system, and high-speed data transfer

VMEbus

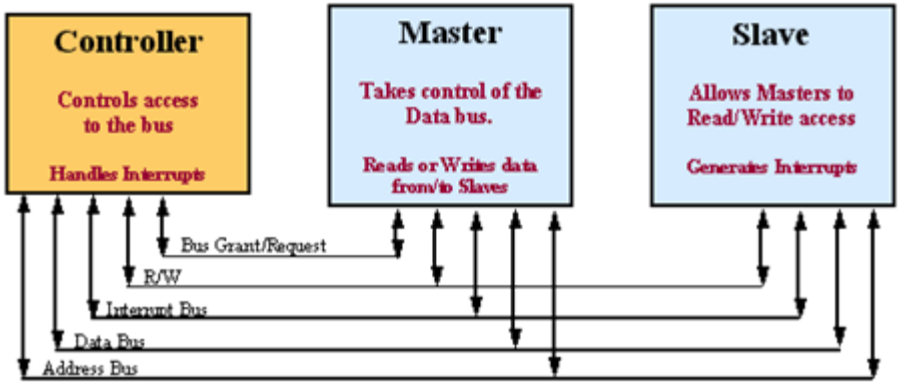
- VMEbus is an international standard. It is used in nearly every HEP experiment to connect computers to detectors, using a mix of commercial and home-made modules
- The standard provides:
  - *Precise mechanical specs*
  - *Precise electrical specs*
- Components – mostly commercial:
  - *Crate providing power & Cooling;*
  - *21 numbered slots*
  - *One or more controllers (one in slot 1)*
    - *Typically a single-board computer*
  - *Slave modules – detector readout, counters,…*
  - *Backplane to interconnect modules*



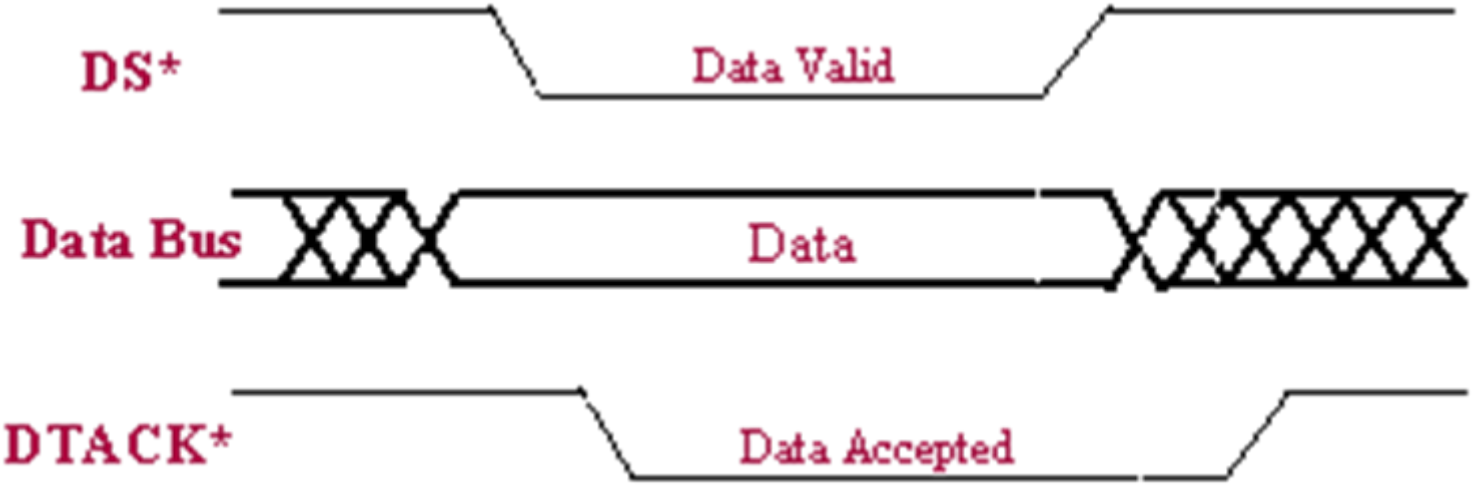
VMEbus – Backplane Signal summary

- Like Motorola 68000 signals, extended into a bus, grown to 32 bits
  - *Handshake protocol – dialogue between Master and Slave*
- Connectors carry the same signals on the same connector pins in each slot
  - *Most signals are bussed; a few are daisy-chained; a few are special*
  - *Signals are +5V or 0V (some other levels introduced recently, not in common use)*
  - *Normally +5V = logic "1", but "*" in a signal name means +5V = logic "0"*
- Data transfer
  - *31 address lines A31-A1(!) + AM5-AM0 + DS0* & DS1* + LWORD*;*
  - *32 data lines D31 – D0;*
  - *AS*, (DS0*, DS1*), BERR*,DTACK*, WRITE* plus others in VME64*
- Arbitration
  - *14 signals. Determine which module becomes master and when*
- Priority interrupts
  - *10 signals*
- Utility bus
  - *SYSCLK*, SYSRESET*, SYSFAIL*, ACFAIL* plus others in VME64*

VMEbus – data transfer



"Timing Diagram"



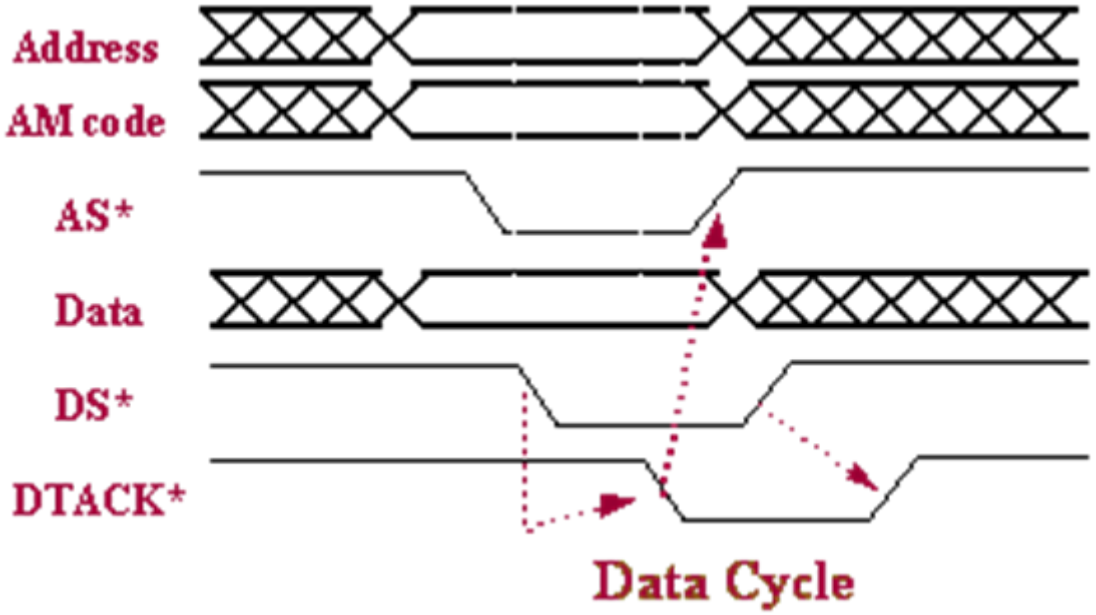
Example: VME Write Cycle (excluding arbitration)

- Master:
- Assert Address, AM Code (cycle type – e.g. 0x09 = A32, D32)
- Assert WRITE*, LWORD*
- Assert AS*
- Assert Data
- Assert DS*

- Slave:
- Inspect Address & AM Code
- Capture Address & Data
- Assert DTACK*

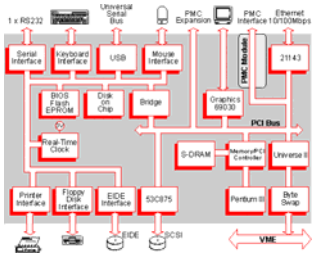
- Master:
- Remove Address, AM & AS*
- Remove Data & DS*

- Slave:
- Remove DTACK*

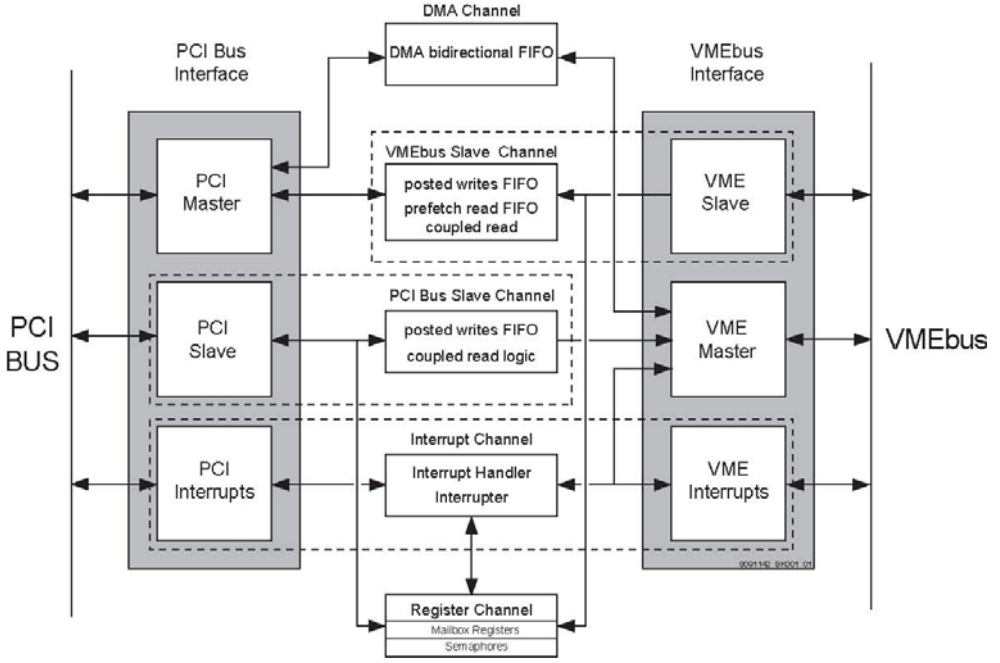


- There are many other types of cycle – e.g. up to A64, up to D64

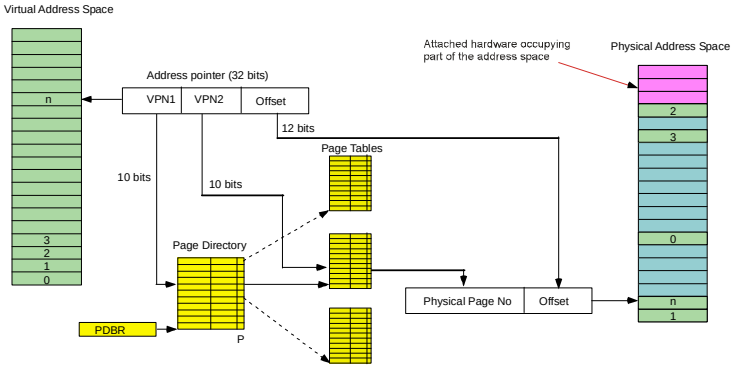
Overview of the Universe
("Tundra Universe", that is)

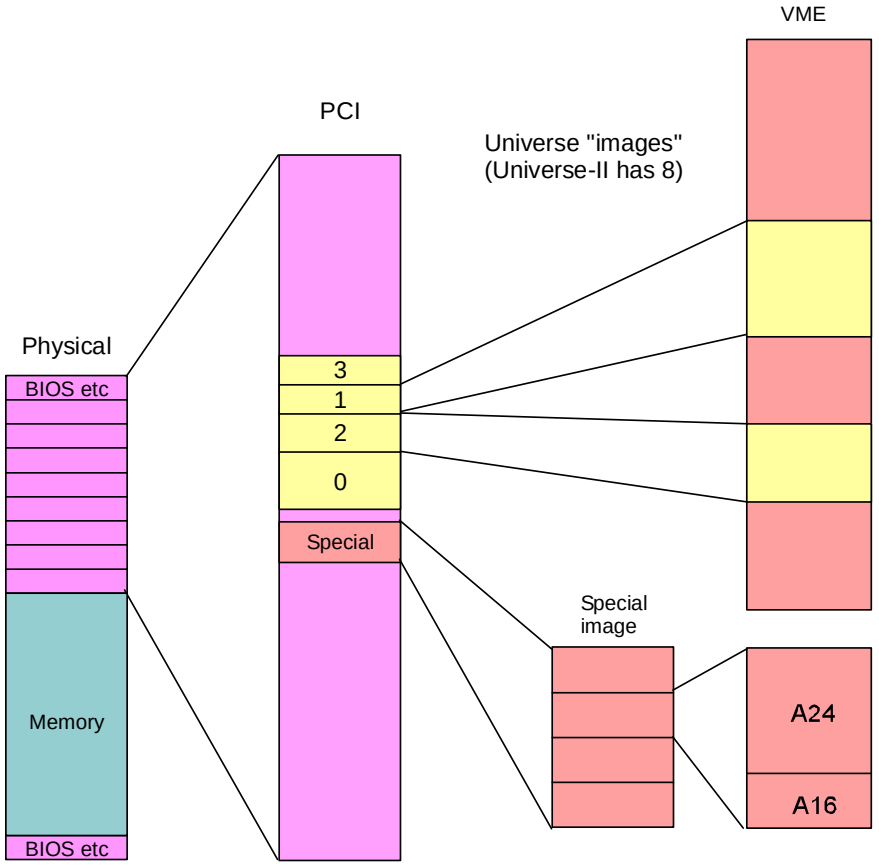


- Manual has 467 pages!!
- Three main sets of functions to support CPU as
  - *a VME Master (computer-initiated cycles)*
    - *The normal case*
  - *a VME Slave (another VME module reads or writes into our computer)*
    - *The other VME module is probably another computer!!*
  - *an Interrupt handler*
    - *Quite often needed*
    - *Very briefly discussed later*



Universe as VME Master





VME addr = PCI addr + offset [31:12].
A16, A24, A32, CR/CSR,..

Image Size, VME_Base and offset, addressing mode all programmable.

VME A24 addr = position in "A24" space

VME A16 addr =position in "A16" space

Summary so far…

- VME is an international standard covering electrical and mechanical details
  - *There are many commercial modules around, institutes commonly make their own as well*
- The protocol involves a handshake between DS(0,1) and DTACK
  - *Every cycle has a defined master and slave*
- Interface hardware maps VME addresses into system physical address space
  - *… looking like memory. An application may need several maps*
- The computer's memory management tables must map the hardware into virtual address space
  - *… so that your software can see it*

- Now your software is talking to the hardware. The next section is about how to be fast

Direct Memory Access - DMA

- Data collection usually involves reading a block of data into computer memory:

  for (i=0; i<ncycles; i++) {
  **destination++ = *source++; }*

  Q: Could I code this to run faster??
- This is CPU intensive (CPU cycles slow down to bus speeds)
- Most PCI interfaces have DMA hardware to do this job faster and autonomously
  - *The CPU sets up the details of the multi-cycle transfer, then starts it*
    - *Then the CPU is free for other work until the transfer completes*
  - *Each individual VME cycle is done by the DMA. Resulting data is written directly to physical computer memory. DMA is PCI AND VME master*
  - *The target VME module can't tell, except that the interval BETWEEN cycles is shorter. The timing INSIDE a cycle is identical*
  - *At the end, an interrupt may be generated, or the software can ask "Finished?*
- What does the DMA hardware need to know?
  - *First source address (in VME); Type of VME cycle to do;*
  - *First destination address (in PHYSICAL memory)*
  - *End Condition (e.g. number of bytes to transfer)*
- The PCI bus is optimised for this sort of traffic – the addresses are sent once per block

Block Transfers

- Many people use the terms DMA and Block Transfer as synonyms
- They are NOT

- DMA is a way of moving data to/from memory without involving the CPU
- Block Transfer (next slides) uses a different type of VME cycle to move data faster
- DMA can use ordinary VME cycles, and does NOT require block transfer.
- Block transfer DOES require DMA

- >> If you specify modules, be clear about what you mean
  - *Making a module DMA-compliant needs no extra work. Making it block-transfer compliant may be quite hard*

VMEbus block transfers (MBLT - new in VME64)

- A fast use of VME, for DMA transfers only.
- Different master-slave contract, indicated by a different AM Code (e.g. 0x08)
- Address sent only once
  - *then incremented internally by Master & Slave after each data transfer*
- Data and address lines all used for data transfer
- No arbitration for bus mastership between cycles

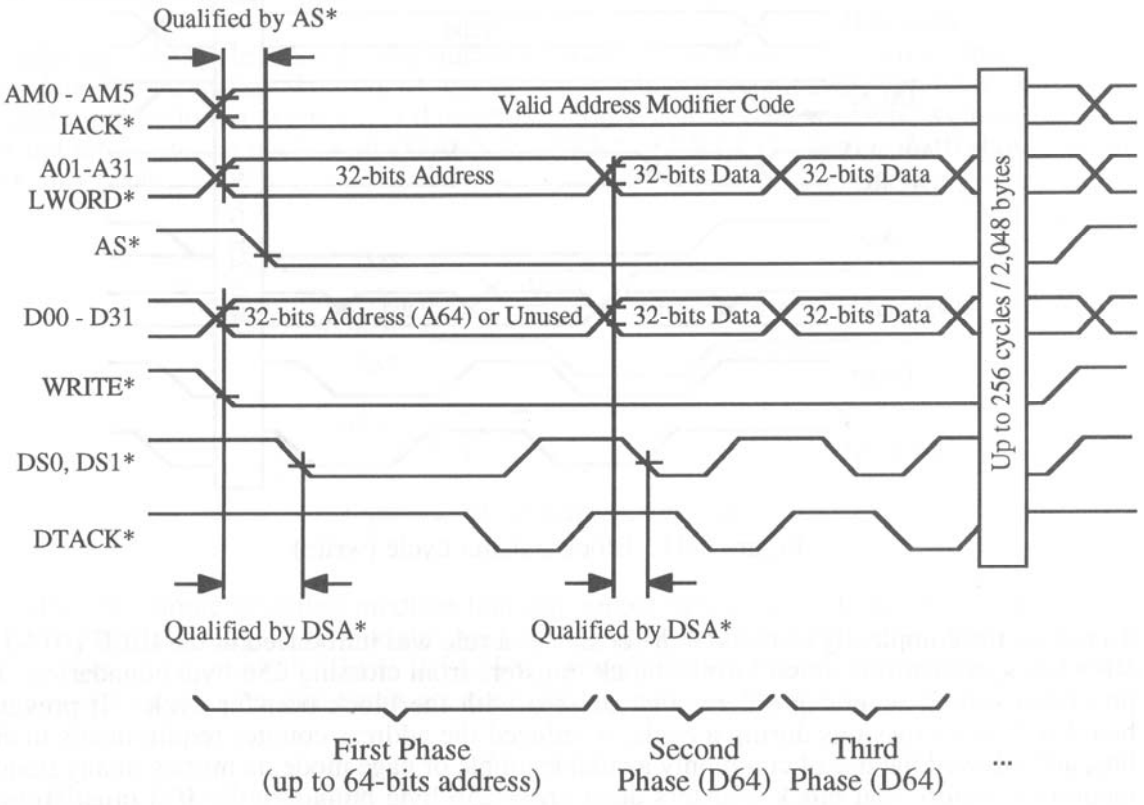


Figure 2-12. MBLT cycle.

Chained DMA

- The DMA paradigm unlocks several tricks to go much faster
- Chained DMA is useful because reading several modules one after another is very common
- Set up a linked list in memory, containing command packets for transactions to be done:
  - *Read 100 words from module 1 (at addr 0x123400, A32, D32, AM=0x09);*
  - *Then 80 words from module 2 (somewhere else in VME space);*
  - *etc. The last command packet has an "end" marker.*
- Each entry in the list has all the info to set up the DMA for one contiguous transfer
- Hand the address of the list to the DMA controller, say GO
  - *The DMA processes the complete list as one transaction*

- During disk swapping and paging, the same idea is used to transfer data between physically discontiguous memory pages and disk blocks

VME64 and extensions (VME64x, VME64xP)

- New cycle types allow for 40- or 64-bit addresses by borrowing some of the data lines
  - *The Address Modifier (AM) codes tell the slave what is happening*
  - *Not all slaves support A40 or A64 addresses*
  - *Of course the data lines can't be used for data during the address part of the cycle*
- 64-bit data transfer ("MBLT") is possible using D32, A31, & LWORD lines
  - *64-bit data can't be sent while the address lines are in use*
- 2eVME transfers capture address or data on each transition of strobes

- 160 Mbytes/sec is claimed for latest VME extensions. I've had ~few Mbyte/sec in programmed cycles, 20 Mbyte/sec in DMA. Getting high speeds needs great care

Summary so far

- There are several ways to make VME go faster.
  - *Being really fast is demanding*
- DMA issues identical VME cycles, but without the CPU
  - *Cycles are more frequent, PCI traffic is in blocks*
- Block transfers use DMA with different VME protocols
  - *Address sent only once on VMEbus. Faster than bare DMA*
  - *Can use wider data in this mode*
- Chained DMA joins a sequence of DMA transfers together as one transaction

- Next: Interrupts and Errors

VME Interrupts (outline)

- Module wants to attract attention of CPU
  - *Some condition satisfied – e.g. front panel input signal from scintillator, count has reached a preset value,…*
- Module asserts one if the 7 interrupt request (IRQ*) lines
- Responding, the SBC becomes bus master
  - *SBC (actually Universe) Issues a "Status/ID" read request (i.e. reads interrupt vector)*
- VME Module responds with interrupt vector (normally 8 bits)
  - *Which the Tundra stores in a PCI-readable register.*
- Tundra generates PCI interrupt
- CPU processes interrupt, calls VME Interrupt handler
- VME interrupt handler reads vector from Tundra
  - *Vector is used to identify target process, to which a Linux signal is sent*
- The application's signal handler should start by clearing the module interrupt

VME interrupt caveats

- The Status/ID read cycle uses the special VME IACK* line
  - *This is a daisy-chain between slots, not a bus*
  - *Jumpers must be installed to bridge empty slots between the interrupting module and the crate master*
    - *Failure to do this will cause a system hang – the interrupt can't be processed or cleared. So put the interrupting module as near to the CPU as possible*
- Think carefully about interrupts vs polling
- To reduce deadtime, prepare the DMA before the interrupt arrives
  - *Then just start it when your event is ready to read out*
- The interrupt vector will be set by jumpers or switches on the module
  - *Make sure your documentation fully describes the procedure to replace a broken module (same slot, copy the jumper settings) or replace a crate (same crate backplane jumper settings)*

Error Detection and Handling

- VME doesn't have any checking that the data and address arrive correctly
  - *But in fact when properly set up, poor data or addresses are very rare*
  - *If you need high-reliability transfers, structure the data accordingly*
    - *e.g. include parity or crc fields in memory blocks*
- A very common error is an attempt to address a missing module
  - *A normal cycle is terminated by the slave generation of DTACK*. If there is no slave, the cycle could continue for ever*
  - *After 64μs, the bus timer will assert BERR* - "Bus Error"*
  - *VME master must now terminate the cycle*
  - *If the master is a computer interface, it will normally raise an interrupt*
  - *Interrupt handler can throw an exception which you can catch*

Bus error handling – code example

Your code:

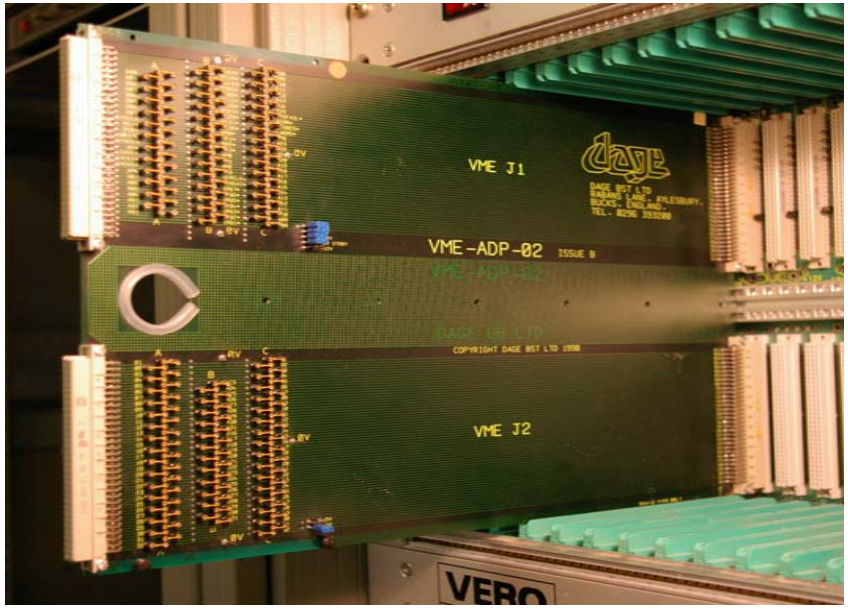
```
long value;
volatile long *myVmeRegister;
try {
        value = *myVmeRegister;
        cout << " I read value: " << value << std::eol;

  }
 catch (BusError& error) {
    std::cout << " Bus error – myVmeRegister doesn't respond." << std:: eol;
  }
```

Practical Hints and Tips

- Take scope shots of signals in the working system so you know want to expect
- Obtain or write a general purpose VME exerciser software tool
  - *It is essential to be able to scan address space for modules, issue single read or write cycles, etc.*
- Use only 16-bit or only 32-bit integers unless other formats are essential
- If you make your own modules
  - *Ask your engineers to add scope test points for VME signals*
    - *At least AS*, DS0*,DS1*, DTACK*, D0, board select.*
  - *Ask for a "board-select" light (to show when you are addressing the module)*
  - *Ask your engineers to obey the VME spec fully – it'll bite you later otherwise*
- In C/C++, declare hardware registers "volatile" to prevent the compiler from optimising them away

Hints and Tips (2)

- Buy a VME extender module – use it to access module internals and VME buys signals for diagnostics – and a VME Bus Display Module

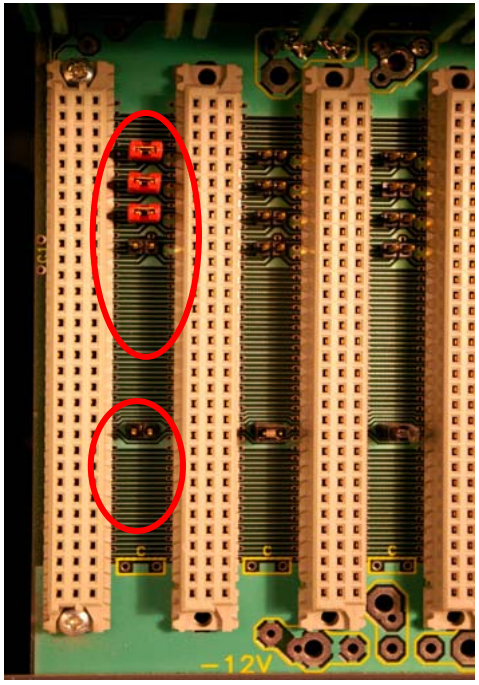


Hints and Tips (3)

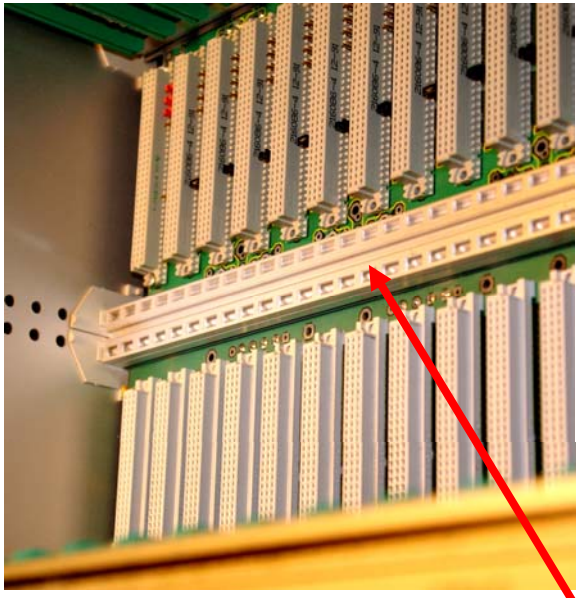
- Be aware of…

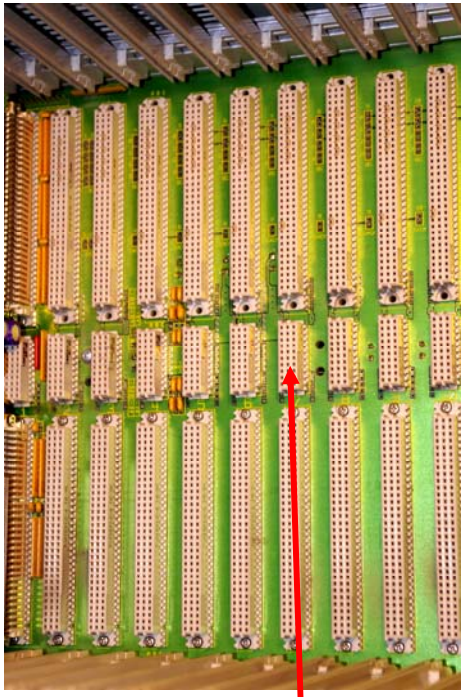
BG 0-3

IACK*



backplane jumpers



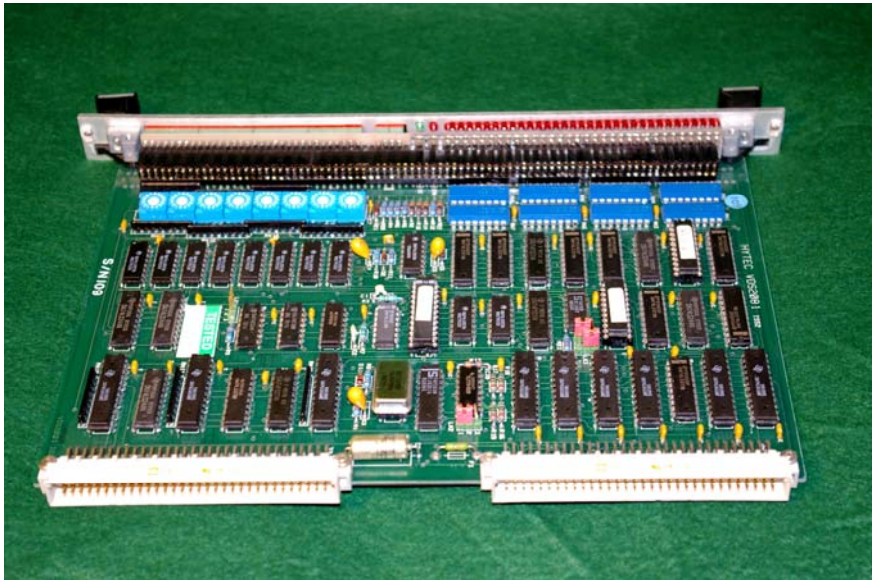


old-style backplanes with bars in the J0 position

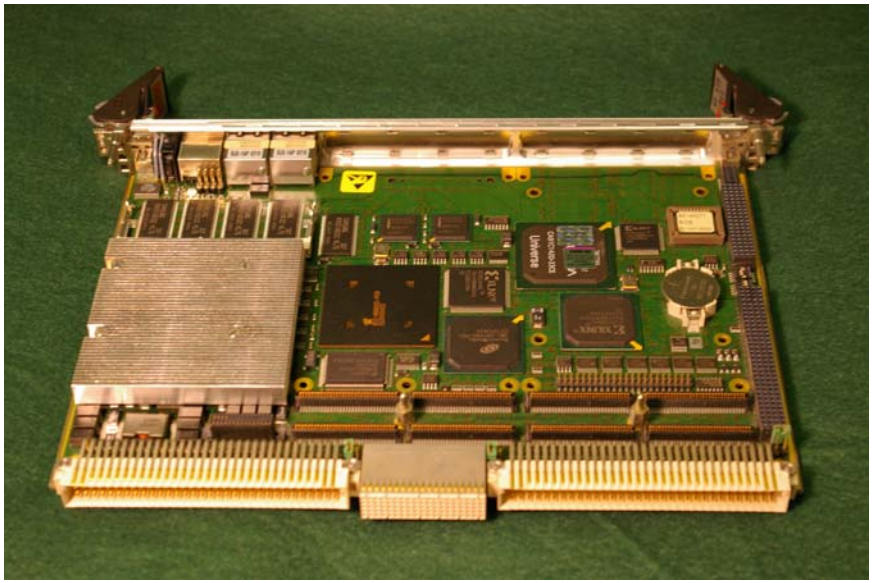
Note: Check if crates and modules use 3-row or 5-row connectors

Hints and Tips (4)

- The old module will fit an old crate backplane. The new module will not.



Old module (VME32)

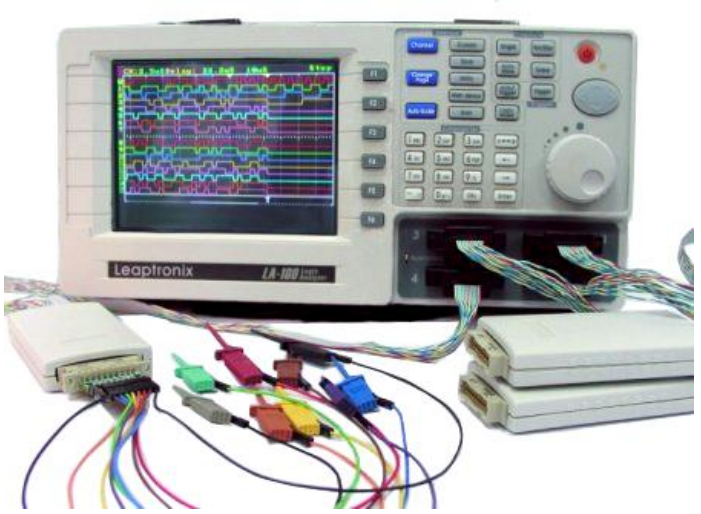


New module (VME64)

- …but both will work in a VME64 crate

Hints and Tips (5)

- Store unused VME modules upside-down in a crate, so they can't be pushed part-way into the backplane
- *** Don't use the front panel screws to pull a module into a crate ***
- Observe antistatic precautions
- On holiday, learn how to use a logic analyser



References

- The VMEbus Handbook, by Wade Peterson, Published by VMEbus International Trade Association (VITA), from https://www.vita.com/online-store.html
- Tundra Universe Specification, at http://www.tundra.com/products.aspx?id=1643
- American National Standard for VME 64 Extensions (i.e. VME64x), ANSI/VITA 1.1-1997, from VITA as above
- American National Standard for VME 64 Extensions for Physics and Other Applications (i.e.VME64xP), ANSI/VITA 23-1998, from VITA as above (as PDF)
- PCI – introductory material in Wikipedia
- RAL has occasionally run a 1-2 day VME course