Lecture 13
Harvard architecture
Coccone OS demonstrator

Computing platforms

Novosibirsk State University
University of Hertfordshire

D. Irtegov, A.Shafarenko

2018

Harvard version of CdM-8

- Harvard architecture = separate memory banks for data and instruction
- Can be implemented as:
  - One-bit address extension selecting code and data access (cheap and simple)
  - Separate memory channels (buses) – expensive but boosts performance
  - Memory buses of different width
    (example: Microchip PIC has 8-bit data and 14-bit instruction memory)
  - Separate cache channels and caches, probably backed by same main memory actually used in some CPU
- In CdM-8, first approach (bit extension) is used
- One ISA-level change: ldc (LoaD Constant) instruction to read data from instruction memory

Coccone: most advanced version of CdM-8

- Extension of CdM-8 architecture and schematics intended to demonstrate basic concepts of protected-memory operating systems
- In second semester we offer team project: actually building OS for this machine
- But what we need to build an operating system?

Processes (tasks)

- Most (but not all) modern operating systems have concept of *process*
- Process is a virtual machine (or a sandbox) with limited access, that runs in isolated memory space
- Process virtual machine is NOT emulating full access to system hardware (unlike hypervisor virtual machines like VMWare or VirtualBox)
- All programs you write in C programming course and most other programs you use (including CocoIDE) are designed to be run as processes and use operating system services to access hardware

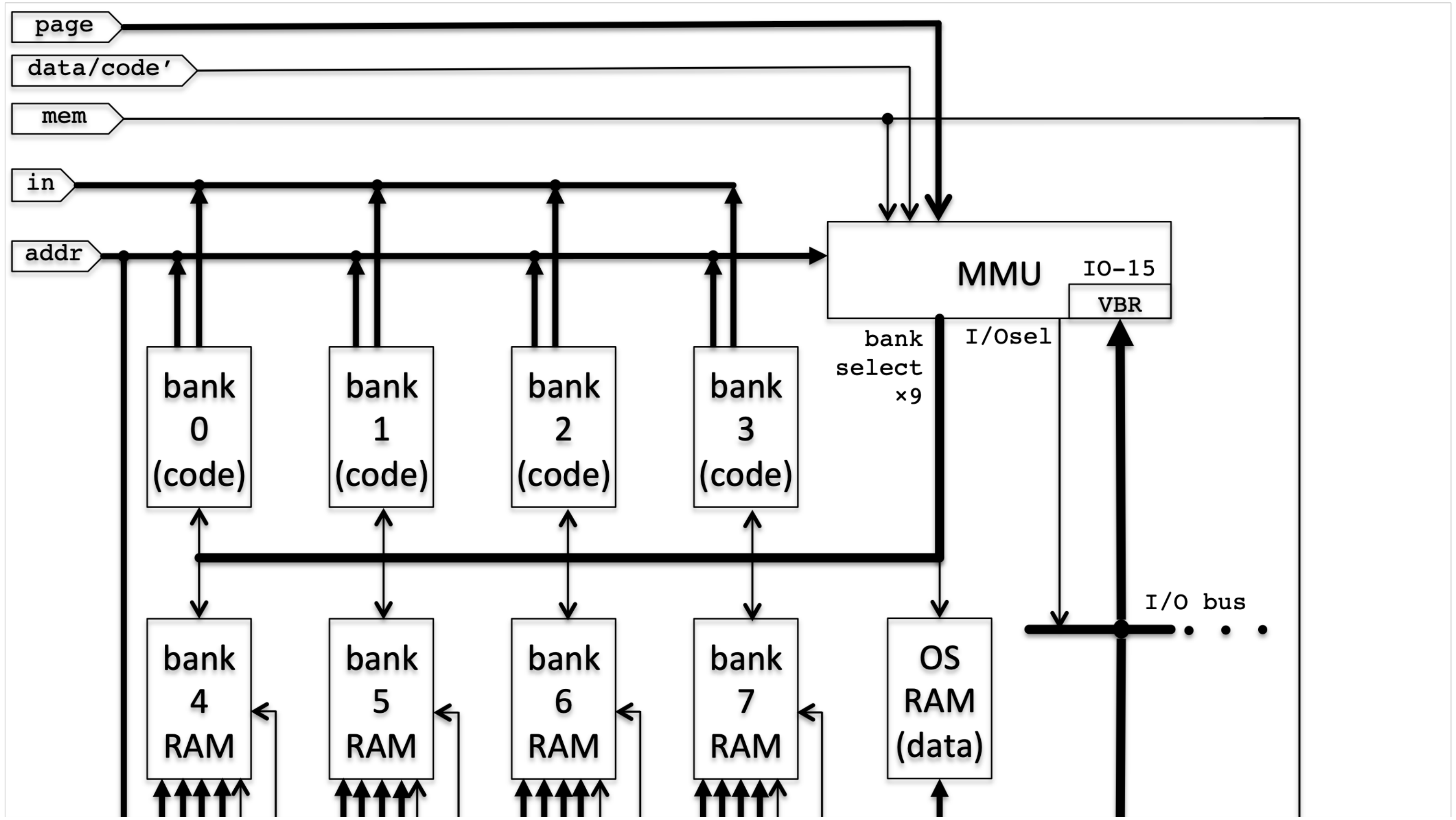
What we need to protect memory

- Essentially, we need to catch every memory access for a running program
- Catch all memory accesses programmatically
  - In fact, we need to interpret entire machine code of your program
  - This is called byte-level interpretation
  - This is how cocoemu actually works
  - Orders of magnitude slower than hardware interpretation
  - Can be significantly sped up by JiT compilation
  - JiT is what Java, C# and many hypervisor virtual machines actually do, but this is far beyond scope of our course

Catch all memory accesses in hardware

- Insert a hardware device (MMU for Memory Management Unit) between CPU and memory bank
- Coccone is using one of simplest known types of MMU, known as memory banks
- Coccone uses 3 previously unused bits in PS register as bank selector
- In tome.pdf and in Logisim schemes it is also referenced as page
- Bits 0-3 - CVZN flags, bit 7 - interrupt enable, bits 4-6 - selector
- So, we can use 8 memory banks 256 bytes each
- Actually, there are 10 banks, but more on this later

Coccone memory subsystem



Memory banks are not equal

- Banks 0-3 are reserved for OS code (even simplest OS won’t fit in single bank)
- When bank 0-3 is active, CPU switches to Harvard mode and can use special 9th bank of RAM for data
- When bank 4-7 is active, CPU switches to Manchester mode and uses the same bank for code and data
- Banks 4-7 are for [user] processes
- Bank 10 is for memory-mapped I/O
- This is controlled by VBR MMU register (byte 0xff of OS data/IO pages)

VBR (Virtual Bank Register)

- Not available in “user mode” (code banks 4-7)
- Can select one of
  - user mode banks,
  - OS data bank
  - I/O page

for data access in “system mode” (code banks 0-3)

`page`	`data/code'`	`addr`	`bank-select`	I/Osel
p	0	any	p	0
$0 \leq p \leq 3$	1	$<$`0xf0`	VBR	0
$0 \leq p \leq 3$	1	$\geq$`0xf0`	none	1
$p \geq 4$	1	any	p	0

How to copy data from OS to user bank?

```
ldi r2,data.KBusr        # memory bank
ld r2,r2                 # r2=memory bank

ldi r0,MMU               #r0-> MMU I/O reg
st r0,r2                 # set data memory bank

st r1,r3                 # set data memory bank
inc r1                   # advance buffer pointer

clr r2                   # MMU reset to page 0
st r0,r2                 #
```

But how to actually switch banks?

- When we write to bank selector, this is indirect jump (PC now points to different bank but to same position in the bank)
- One solution: place a same piece of code in every bank
- This code will handle bank switching
- This approach is used in many 8-bit CPU with memory banks
- Actually, this is used in many OS for 32- and 64-bit CPUs. A
- All OS for x86 are using this approach (OS kernel is mapped to same addresses in all processes)

And how actually change bits 4-6 of PS?

- We know only two instructions that load and store full PS register: *ioi* and *rti*
- We can use *rti* to put arbitrary value in PS (*push* it and then *rti*)
- But *rti* is also a control transfer (considering a previous slide, this is good!)
- And you can use ioi to call procedures in other banks (just place right PS value at vector 0!)
- (Actually, many OSes use software interrupts for system calls)
- Also, special instruction osix with single operand (equivalent to ioi to vector 0, but bits 0-6 of new PS are taken from the operand)

But how interrupts work in multibank system?

- Where interrupt vectors are placed?
- What happens to the stack?
- Coccone always takes interrupt vectors from bank 0
- But the vector contains a bank selector, so the handler can be placed in different bank!
- Coccone stack pointer is *shadowed*
- There are actually 8 stack pointers, one for every bank
==one per process
- But during interrupts, SP[0] is always used, so placing ISR in other banks require extra work

More on osix and rti

- *osix* can select target bank and set flags in PS
- Handler in the bank can use flags and conditional branches to decode syscall number
- We can have 4x16=64 useful syscalls (pointing to OS code pages)
- And also 64 useless syscalls pointing to user code pages
  - Useless syscalls actually disturb system operation, so coccone is not as protected as “real” protected memory systems
- *rti* can put arbitrary values to PS on return
- Syscalls can use flags to indicate failure or success
- Syscalls can pass parameters in registers
- We’ve seen how syscalls can copy data to user space, so we can pass pointers as parameters