


21-Nov-06 (1)




CSC2510 - Computer Organization

Lecture 9: Memory and Basic Processor Unit

Philip Leong


21-Nov-06 (4)



Cache

- Read request
 - Contents of a **block** are read into the cache the first time
 - Subsequent accesses are (hopefully) from the cache (called a **read hit**)
 - Number of cache entries is relatively small, need to keep most likely data in cache
 - When an uncached block is required, need to employ a **replacement algorithm** to remove an old block and to create space for the new
- Write operation
- Scheme 1
 - Cache and memory updated at the same time (**write-through**)
- Scheme 2
 - Update cache location and mark as **dirty**. Main memory updated when cache block is removed (**write-back**)
- Which is simpler? Which has the better performance? Why?
- Read misses, read hits, write misses, write hits can occur
- **Mapping functions** determine how addresses are assigned to cache locations

21-Nov-06 (2)




Midterm

- Everybody did well!
- Q1d

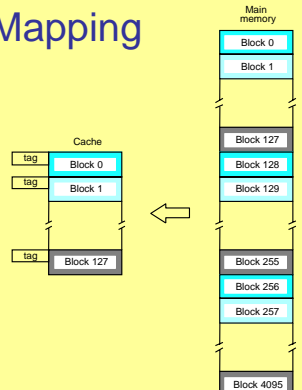
Address	Instruction	ADDRESS
1000	Move X,-(SP)	1024
1004	Move Y,-(SP)	1024
1008	Call A	1012
..		Y
1020	A Call B ; POSITION 2	X
1024	Add #4,SP ; POSITION 3	
1028	Return	
1032		
1036	B Move (SP),R1	
1038	Move R1,-(SP) ; POSITION 1	
1040	Return	

21-Nov-06 (5)



Direct Mapping


- Memory address divided into 3 fields
 - Block determines position of block in cache
 - **Tag** used to keep track of which block is in cache (as many blocks can map to same position in cache)
 - Word selects which word of the block to return for a read operation
- Given an address t,b,w
 - See if it is already in cache by comparing t with the tag in block b
 - If not, replace the current block at b with a new one from memory



Tag	Block	Word
5	7	4

Main memory address

21-Nov-06 (3)




Midterm

- Q2b
- Q2c

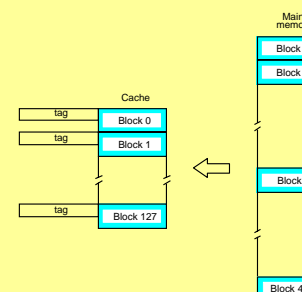
a	dd	0,0,0,0	mov	eax,0	
			1	cmp	ecx,0
	lea	ebx,a		je	fin
	mov	0[bx],2		add	eax,ecx
	mov	4[bx],1		dec	ecx
	mov	12[bx],0		jmp	l
				fin	...

21-Nov-06 (6)



Associative Mapping

- In direct mapping, the block is restricted to reside in a given position in the cache
- Associative mapping allows block to reside in an arbitrary cache location
- In this example, all 128 tag entries must be compared with the address Tag in parallel



Tag	Word
12	4

Main memory address

Set Associative Mapping

- Combination of direct and associative
- Blocks 0,64,128,...,4032 map to cache set 0 and can occupy either of 2 positions within that set
- A cache with k-blocks per set is called a **k-way set associative cache**

Example

Memory address	Contents
(7A00)	A(0,0)
(7A01)	A(1,0)
(7A02)	A(2,0)
(7A03)	A(3,0)
(7A04)	A(0,1)
...	...
(7A24)	A(0,9)
(7A25)	A(1,9)
(7A26)	A(2,9)
(7A27)	A(3,9)

Tag for direct mapped (bits 0-5)
 Tag for set-associative (bits 0-11)
 Tag for associative (bits 0-11)

Replacement Algorithms

- Direct mapped cache
 - Position of each block fixed, no replacement algorithm needed
- Associative and Set associative
 - Need to decide which block to replace (keep ones likely to be used in cache)
 - One strategy is **least recently used (LRU)** e.g. for a 4 block cache, use a 2-bit counter. Set =0 for block accessed, other blocks incremented. Block with count=3 replaced upon miss
 - Another is **random replacement** (choose random block). Advantage is that it is easier to implement at high speed

Direct Mapped

- Least significant 3-bits of address determine location in cache
- When i=9 and 8, get a cache hit (2 hits in total)
- For j loop only 2 out of the 8 cache positions used
- Tags not shown but are needed
- Very inefficient cache utilisation

Block position	j = 1	j = 3	j = 5	j = 7	j = 9	i = 6	i = 4	i = 2	i = 0
0	A(0,0)	A(0,2)	A(0,4)	A(0,6)	A(0,8)	A(0,6)	A(0,4)	A(0,2)	A(0,0)
1									
2									
3									
4	A(0,1)	A(0,3)	A(0,5)	A(0,7)	A(0,9)	A(0,7)	A(0,5)	A(0,3)	A(0,1)
5									
6									
7									

Example

- Assume separate instruction and data caches
- Cache has space for 8 blocks
- Block contains 1 16-bit word
- A(4,10) is an array of words located at 7A00-7A27 in column order
- Normalise an array by its average

```

SUM := 0
for j:= 0 to 9 do
    SUM := SUM + A(0,j)
end
AVE := SUM / 10
for i:= 9 downto 0 do
    A(0,i) := A(0,i) / AVE
end
    
```

Associative-mapped

- LRU replacement policy
- Get hits for i=9,8,...,2
- If i loop was a forward one, would get no hits!

Block position	j = 7	j = 8	j = 9	i = 1	i = 0
0	A(0,0)	A(0,8)	A(0,8)	A(0,8)	A(0,0)
1	A(0,1)	A(0,1)	A(0,9)	A(0,1)	A(0,1)
2	A(0,2)	A(0,2)	A(0,2)	A(0,2)	A(0,2)
3	A(0,3)	A(0,3)	A(0,3)	A(0,3)	A(0,3)
4	A(0,4)	A(0,4)	A(0,4)	A(0,4)	A(0,4)
5	A(0,5)	A(0,5)	A(0,5)	A(0,5)	A(0,5)
6	A(0,6)	A(0,6)	A(0,6)	A(0,6)	A(0,6)
7	A(0,7)	A(0,7)	A(0,7)	A(0,7)	A(0,7)

21-Nov-06 (13)



Set associative

- Since all accessed blocks have even addresses, only half the cache is used i.e. they all map to set 1
- Get hits for $i=9,8,7,6$
- Random replacement would have better average performance

Contents of data cache after pass:

	$j = 3$	$j = 7$	$j = 9$	$i = 4$	$i = 2$	$i = 0$
Set 0	A(0,0)	A(0,4)	A(0,8)	A(0,4)	A(0,4)	A(0,0)
	A(0,1)	A(0,5)	A(0,9)	A(0,5)	A(0,5)	A(0,1)
	A(0,2)	A(0,6)	A(0,6)	A(0,6)	A(0,2)	A(0,2)
	A(0,3)	A(0,7)	A(0,7)	A(0,7)	A(0,3)	A(0,3)
Set 1						

21-Nov-06 (16)



Intel Core 2 Duo

- Number of processors 1
- Number of cores 2 per processor
- Number of threads 2 (max 2) per processor
- Name Intel Core 2 Duo E6600
- Code Name Conroe
- Specification Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz
- Technology 65 nm
- Core Speed 2843.6 MHz
- Multiplier x Bus speed 9.0 x 316.0 MHz
- Rated Bus speed 1263.8 MHz
- Stock frequency 2400 MHz
- Instruction sets MMX, SSE, SSE2, SSE3, SSSE3, EM64T
- L1 Data cache 2 x 32 KBytes, 8-way set associative, 64-byte line size
- L1 Instruction cache 2 x 32 KBytes, 8-way set associative, 64-byte line size
- L2 cache 4096 KBytes, 16-way set associative, 64-byte line size

21-Nov-06 (14)



Comments on example

- In this example, associative is best, then set associative and direct
- What are the advantages and disadvantages of each scheme?
- In practice, low hit rates like in the example is very rare
- Larger blocks and more blocks greatly improve cache hit rate

21-Nov-06 (17)



Gcc parameter passing

- In C sieve(int n, char a[])
 - Assuming n is in %ecx and a[] is in %ebx, called using
- ```

pushl %ebx
pushl %ecx
call _sieve

.file "sieve.c"
.text
.globl _sieve
.def _sieve; .scl 2;
.type 32; .endef
_sieve:
pushl %ebp
movl %esp, %ebp
... ; save registers etc
movl 12(%ebp), %edi ; a
movl 8(%ebp), %eax ; n

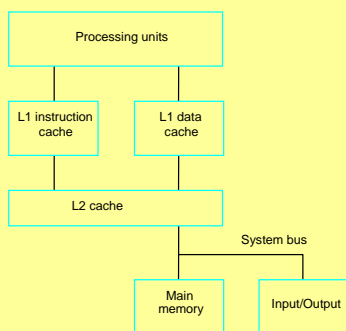
...
popl %ebp
ret

```

21-Nov-06 (15)



## Intel Core 2 Duo

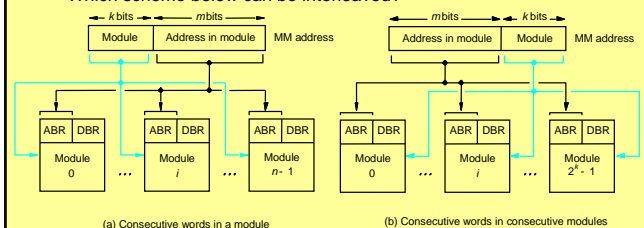


21-Nov-06 (18)



## Interleaving

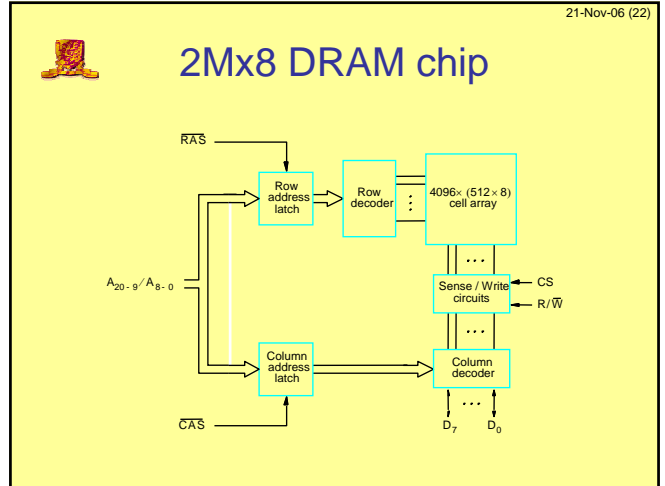
- Processor is fast, memory slow. Try to hide access latency by interleaving memory accesses across several memory modules. Each module has own address and data buffer register (ABR & DBR)
- Which scheme below can be interleaved?



21-Nov-06 (19)

## Example

- Suppose we have a read miss and need to load from main memory
- Have cache with 8-word block
- Assume takes one clock to send address to DRAM memory and one to send back to cache. In addition DRAM has 8 cycle latency for first word but subsequent words in same row take 4 cycles
- Single memory  $1+8+(7 \times 4)+1=38$  cycles
- For interleaved scheme, send addresses, 2 accesses needed from the 4 memories and then 4 reads  $1+8+4+4=17$  cycles. Transfer of first 4 words are overlapped with access of second 4 words



21-Nov-06 (20)

## Single memory

Cycles:  $1+8+7 \times 4+1$

21-Nov-06 (23)

## Hit rate and Miss penalty

- Goal is to have a memory system with speed of cache and size of a hard disk
- High hit rates ( $> 90\%$ ) are essential
  - Miss penalty must also be reduced
- Example
  - h is hit rate
  - M is miss penalty
  - C is cache access time
  - Average access time =  $hC + (1-h)M$
- Example: Assume DRAM memory, 10 cycles for read, 8-word blocks, interleaved memory, then 17 cycles needed to load a block from memory (previous example)
- 30% of instructions are memory read/write (i.e. 130 memory accesses for every 100 instructions), hit rate = 0.95 for instructions and 0.9 for data
- Time without cache =  $130 \times 10$
- Time with cache =  $100(0.95 \times 1 + 0.05 \times 17) + 30(0.9 \times 1 + 0.1 \times 17)$
- Ratio = 5.04 i.e. cache speeds execution 5x

IN MODERN MACHINES, COST OF LOADING BLOCK FROM MEMORY EVEN HIGHER SO RATIO EVEN LARGER

21-Nov-06 (21)

## Interleaved

Cycles:  $1+8+4+4$

21-Nov-06 (24)

## Virtual Memory

- Physical memory not as large as address space spanned by processor
  - E.g. processor might address 32-bits (4G) but memory may only be 1G
- Part of program not in main memory stored on hard disk and brought into main memory (using DMA) as needed
- This is done automatically by the OS, application program does not need to be aware of the existence of **virtual memory (VM)**
- **Memory management unit (MMU)** translates **virtual addresses** to **physical addresses**

21-Nov-06 (25)



## Address Translation

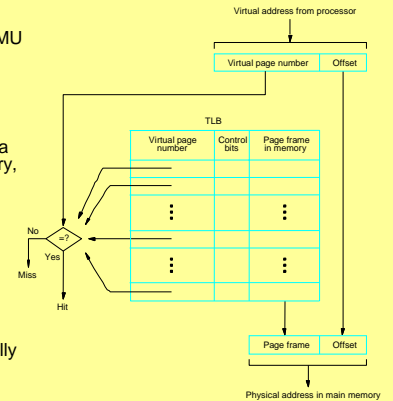
- Memory divided into **pages** ranging from 2K to 16K
  - Too small and too much time will be spent getting pages from disk
  - Too large and a large portion of the page may not be needed
  - C.f. cache block size
- Processor generates virtual addresses. High bits are the **virtual page number** and low bits are the **offset**
- Information about where each page is stored is maintained in a software controlled data structure in main memory called the **page table**
  - Starting address of page table called the **page table base register**
  - Address in physical memory obtained by indexing the virtual page number from the page table base register

21-Nov-06 (28)



## TLB

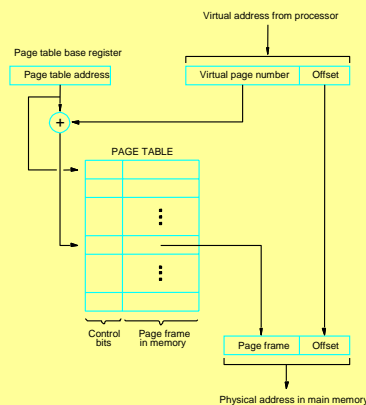
- Given virtual address, MMU first looks in TLB, if available, uses it. Otherwise, obtains PTE from main memory and updates TLB
- When program requests a page that is not in memory, MMU generates a page fault
- Process suspended and control given to OS
- OS must load page from disk into memory
  - Takes a long time, OS may schedule another process to run
- Original process eventually restarted by reexecuting instruction



21-Nov-06 (26)



## VM Address Translation



21-Nov-06 (29)



## Some details

- If memory full, need page replacement algorithm e.g. LRU
  - Usually involves marking pages when they are used and clearing this bit periodically
- Write-through scheme is too inefficient
  - Normally, dirty pages are written back to disk only when they are replaced
- A separate page table is usually maintained for each program (**user space**)
- OS runs in a **system space**
- Processor has 2 states, **system** and **user**
- In user state, some **privileged instructions** cannot be executed e.g. user cannot modify the page table or page table base register

21-Nov-06 (27)



## Page Table

- Page table entries (PTE) also include
  - Control bits describing status of page e.g. whether page is actually in memory, whether the page has been modified, does program have read permission, does program have write permission
- Page table access is required every memory access, to speed it up, keep a cache of PTEs in the **translation lookaside buffer (TLB)**
  - Associative or set associative scheme normally used
  - Processor must keep TLB and page table information consistent

21-Nov-06 (30)



## Basic Processing Unit (Ch 7)

21-Nov-06 (31)

## Fundamental Concepts

- Program execution
- 1. Fetch.  $IR \leftarrow [[PC]]$ . Increment PC.  $PC \leftarrow PC+4$
- 2. Execute actions specified by instruction in IR
- Memory bus controlled through MAR, MDR
- Registers include Rx, Y, Z and TEMP
- MUX selects one of 2 inputs
- Black parts are **datapath**
- Blue parts are **control**

21-Nov-06 (34)

## 2 – ALU or Logic Operation

- ALU is circuit with no clock, apply inputs, wait for its propagation delay, and output appears
- $R3 \leftarrow R1+R2$ :
  - R1out, Yin
  - R2out, Select Y, Add, Zin
  - Zout, R3in

21-Nov-06 (32)

## Execute

- Involves sequence of steps
- Based on the following primitives
  - Transfer word of data from register to another or the ALU
  - Perform arithmetic or logical operation and store result in a register
  - Fetch contents of a memory location to a register
  - Store register to a memory location

21-Nov-06 (35)

## Fetching word from memory

- MDR, MAR connections to bus
- 1. R1out, MARin, Read
- 2. MDRinE, WMFC (wait for MFC)
- 3. MDRout, R2in

For Move (R1),R2

- $MAR \leftarrow [R1]$
- Start read on memory bus
- Wait for Memory Function Completed (MFC) from memory
- Load MDR from bus
- $R2 \leftarrow [MDR]$

Figure 7.4. Connection and control signals for register MDR.

21-Nov-06 (33)

## 1 - Register transfer

- Input and output of register Ri controlled via switches Riin and Riout
- $R4 \leftarrow R1$ : Set R1out to 1 and R4in to 1 (others all 0)
- All transfers synchronised to a **processor clock**
- Riout called **tristate buffer**

21-Nov-06 (36)

## Read timing

- R1out, MARin, Read
- MDRinE, WMFC (wait for MFC)
- MDRout, R2in

21-Nov-06 (37)



## Storing word in memory

- Move R2,(R1) is similar to the previous example
  1. R1out, MARin
  2. R2out, MDRin, Write
  3. MDRoutE, WMFC