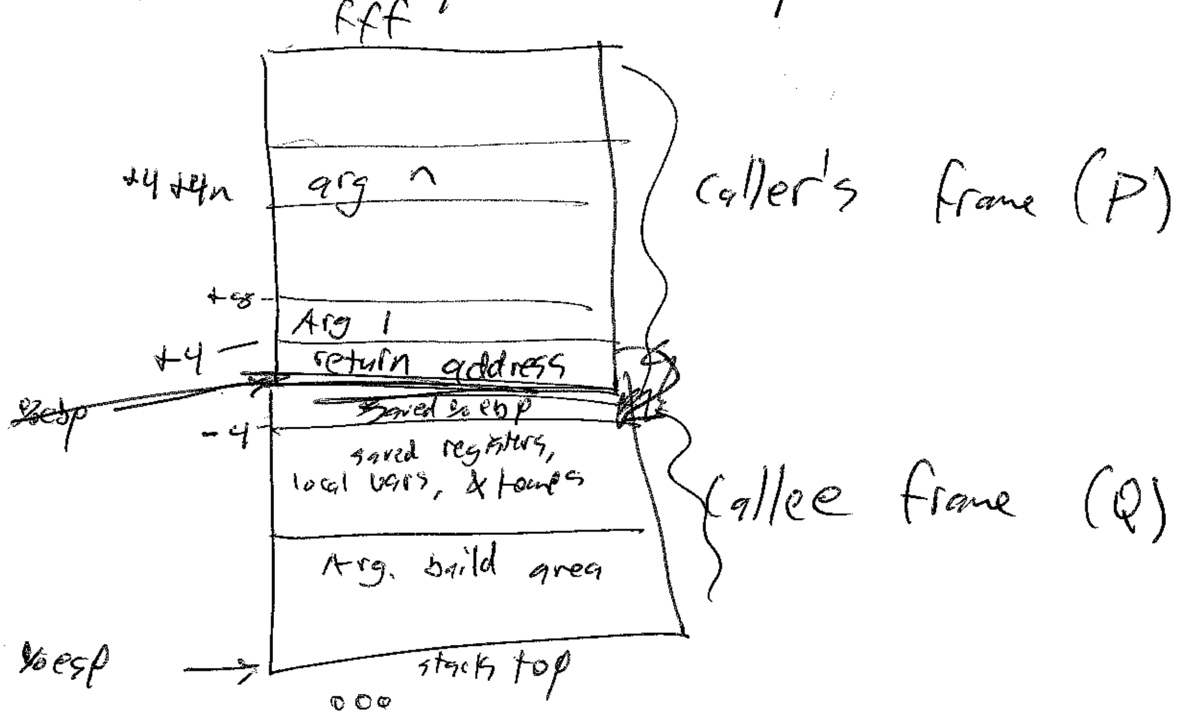
Textbook Notes 3.7-

Procedures

The portion of stack allocated for 1 procedure = stack frame

- %ebp = frame pointer at bottom
- %esp = stack pointer at top



Q also uses the stack for local vars that can't be saved in registers. AKA if:

- not enough registers for all local dat
- some local vars are arrays or structures
- the address operator & is applied to a local var (so we need to generate an address for it)

Register use conventions:

- %eax, %edx, and %ecx = caller save
  ⇒ no need to worry about overwriting them
- %ebx, %esi, & %edi = callee-save
  ⇒ if you fux wit dem, save 'em first.

So if you're about to call some procedure & you have a val (y) you need after, you can

1. store y in your stack frame (caller save) OR
2. store y in a callee-save register so you know it'll be there after.

- Note GCC always allocates a multiple of 16-bytes for a frame (including 4 bytes for old %ebp & 4 bytes for ret address).

Recursion:

- these conventions ⇒ recursion = possible

```
rfact:   #int rfact(int n) ⇒ n!
   pushl  %ebp
   movl   %esp, %ebp                                     } set up
   pushl  %ebx          # callee save
   subl   $4, %esp      # allocate 4 bytes to stack

   movl   8(%ebp), %ebx   # get n
   movl   $1, %eax        # base return = 1              } base or recurse
   cmpl   $1, %ebx        #
   jle    base            # IS base case?

   leal   -1(%ebx), %eax   # compute n-1
   movl   %eax, (%esp)     # put n-1 into top of stack   } recurse
   call   rfact
   imull  %ebx, %eax
```

Textbook Notes 3.7 (p.2) & 3.8

```
addl  $4  %esp  }
popl  %ebx      } wrap-up
popl  %ebp      }
ret
```

Array Allocation & Access:

For data type T & integer constant N,

T A[N];

=> 1.) allocates $L \cdot N$ contiguous bytes in mem.

L = size (in bytes) of T

AND 2.) Makes $A \leftarrow x_A$ = pointer to first elem. in arr

-Accessing $A[i] = x_A + L*i$, $i \in [0, N-1]$

-There are memory addressing tricks for arrays in IA32

e.g.: %edx ← E, %ecx ← i, and E = array of ints

good practice => movl (%edx, %ecx, 4), %eax

=> move E[i] into %eax

Pointer arithmetic

$*(A+i) = A[i]$

=>

Nested Arrays:

int A[5][3];

=

typedef int row3_t[3];

row3_t A[5];

Fixed-Size arrays:

```
# define N 16
typedef int fix_matrix[N][N];
```

⇒ many assembly-level optimizations.

Variable-Size arrays:

- we can do variable-size arrays

```
int val_ele(int n, int A[n][n], int i, int j){
    return A[i][j]; }
```

Textbook Notes 3.9 Heterogeneous Data Structures

Structures:

- struct creates a data type that's a group of possibly different data types.
- compiler knows the byte offset of each field
- selection of field occurs solely @ compile time. machine code has no knowledge of data structures

Unions:

- allows single objects to be referenced according to multiple types.

```
union U3 {
    char c;
    int i[2];
    double v;
};
```

⇒

	c	i	v	size
S3	0	4	12	20
U3	0	0	0	8

refer to the same memory block. i.e. U3 can have one of the above fields.

- Good for saving space, but can lead to nasty bugs. e.g. of data type with mutually exclusive fields: (bin. tree)

```
union NODE_U {
  struct {
      union NODE_U *left;
      union NODE_U *right;
   } internal;
   double data;
};
```

You can do some cool motherfucking things w/ unions

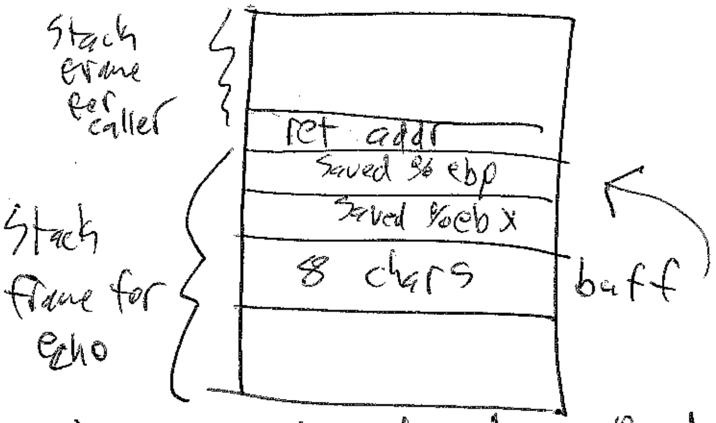
- Some systems place restrictions on addressing (requiring each address start at a multiple of 2, 4, or 8).
- Also data alignment ⇒ major improvement.
- Malloc has to return a pointer that'll work on the worst-case machine (required 4 or 8 alignments)

3.12 Out-of-Bounds mem. references & Buffer Overflow

- Since C doesn't bound-check pointers & local vars are stored on the stack w/ state info, we can get bad program errors

eg. Buffer overflow:

- ~~string~~char array allocated on stack, but size of string exceeds space allocated for the array



7⁺ chars ⇒ overwrites

⇒ bad %edx | Bad %ebp = can't reference local vars or params | bad ret addr = jumps to unexpected location in the program.

Textbook notes

3.12 p. 2

- gets, strcpy, strcat, & sprintf can all generate a byte sequence w/o being given any indication of the size of destination buffer
- Or you can feed the program a string w/ the byte encoding of some executable code (exploit code) + some extra bytes that overwrite the return address w/ a pointer to the exploit code.
  - eg call to start up a shell program
  - <u>or</u> do something, restore code, and return like normal...
- worm = prog. that can run on its own & propogate to other machines
- virus = add on to other programs (including OSs).

Defense against the dark Arts:

1. Stack Randomization

- changing the memory location of the stack from one program ran to another
- Before it was uniform ⇒ one attack works on many machines ⇒ security monoculture.
- works by allocating random amount of space 0-n bytes on the stack at the start of a program.
  - eg by alloca
  - ⇒ this space not used by the program, but all locations are offset by ~~math~~ #.
  - eg test w/ my OS

- This has become standard practice in linux systems
  - part of a larger class of techniques: address-space layout randomization
  - ⇒ entirely different mapping of memory locations each running
- But, ~~de~~ attacker can overcome by brute force attempts w/ different addrs.
  - A trick is to include a long sequence of "nops" before the exploit code (AKA a "nop sled")
  - ⇒ if the prog lands anywhere in there it just slides to the exploit code

2. Stack Corruption Detection:

- Include a randomly generated "canary" b/t local buffer & the rest of the stack state.
- Before restoring register state, verify canary ~~val~~ val is what it should be. If not, abort w/ error.
- GCC does this by default but can be overridden

3. Limiting Executable Code Regions

- ~~[illegible]~~ don't give execute permissions to the stack
- But some programs have to dynamically generate & execute code. eg ~~Java~~ "just-in-time" compilation techniques for interpreted languages (like java)

Textbook Notes 6.0

- In reality, memory systems = hierarchy of storage devices w/ different capacities, costs, & and access times.
  - CPU Registers hold most frequently used memory
  - cache memories = small, fast, nearby CPU, staging areas for some memory
  - Main mem = large, slow disks.
  - external mem = disks or tapes of other machines connected by networks.
- Registers = 0 cycles
- cache = 1 - 30 cycles
- main mem = 50 - 200 cycles
- disk / external = 10's of millions of cycles

- ⇨ write programs w/ good locality!
  - frequently use closer memory

6.1.4 Storage technology Trends

- Price & performance trade-offs

  SRAM faster than DRAM faster than disk

  SRAM costs more than DRAM costs more than disk

- But price & performance properties of different storage technologies are changing at very different rates.
  - since 1980, cost & performance of SRAM have improved at about the same rate.
    - access ↓ by factor of 200
    - cost / megabyte ↓ by factor of 300

- But not so for DRAM
    - cost/meg ↓ by factor of 130,000!
    - access time ↓ by factor of only 10...
- Disk = even more so!
    - cost/meg ↓ by factor of 1,000,000
    - access time ↓ by only factor of 30...
- CPU cycle time ↑ by 2,500 x
    - effective cycle time (divided by # of ~~core~~ processor cores) ↑ by 10,000 x
- Gap b/t CPU performance & DRAM performance is widening.

Text book Notes 6.2 Locality

- Locality = when programs reference data items that are near other recently referenced data items, or that were recently refrenced themselves

  ⇒ HUGE impact on design & peformance

- temporal locality = if it's ref.d once, it's likely to be ref.d again multiple times in near future.
- spatial locality = if it's ref'd, nearby memory will likely be ref'd soon
- Hardware locality = cached memory

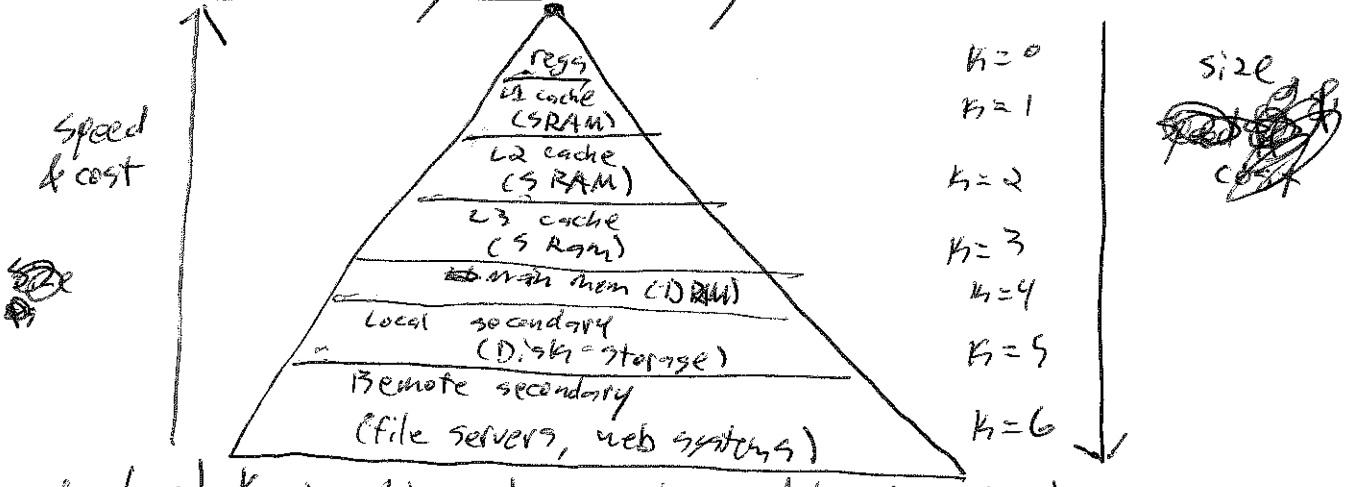
<u>Locality in data ~~reference~~ access:</u>

- Consider looping through each elem. in array for summing
  - this is "stride-1 reference pattern" or "sequential ref."
  - skipping to every k^{th} elem. ⇒ stride-k reference pattern.
  - ↑ k ⇒ ↓ spacial locality
- This is <u>big</u> for multi-dimensional arrays
  - access all elems. of 1 row then next row ⇒ good. Stride-1
  - AKA row-major order

<u>Locality of instruction fetches:</u>

- For loop = good spatial locality
  - 'Cause instr.s held in sequential memory blocks
- Also good temporal locality
  - 'Cause instr.s ~~are~~ repeated.
- Smaller loop body + ↑ iterations ⇒ ↑ locality

6.3 The memory Hierarchy:



- each level k in hierarchy caches data for level k+1
  - contains a subset of (usually) fixed size data blocks from k+1
- block size for L1→L0 usually = 1 byte
- for L2→L1, L3→L2, & L4→L3 usually ≈ 8-16 bytes
- for L5→L4 ~ 100's or 1000's of bytes

- cache hit = when we need d from level k+1 & it's stored in level k
- if not, Cache miss. ⇒ k fetches & stores the data from k+1
  - possibly overwriting a block if cache is full.

- empty cache = cold cache. ⇒ compulsory / cold misses.
- when there's a miss, cache needs a placement policy. (where to place block)
  - most flexible = let the blocks be stored anywhere.
    - ↑ this = too expensive for high-level caches
    - ⇒ restrictive placement policies
- but restrictive policies ⇒ conflict miss.
- Also, capacity misses, when each phase has some reasonably constant set of cacheblocks (the working set) that's too big to fit in the cache.

Textbook Notes 6.3 ctd & 6.4

- This is why temporal & spatial locality work.
  temporal 'cause recent access → it's still in cache
  spatial 'cause if it's close in mem, the block might include it in cache.

6.4 Cache memories

- L1 cache ~ 2-4 clock cycles
- as gap b/t CPU & main mem. grows, more levels of clks
  ⇒ L2 ~ 10 clocks, L3 ~ 30-40 clock cycles

consider computer system w/ m-bit memory addresses.

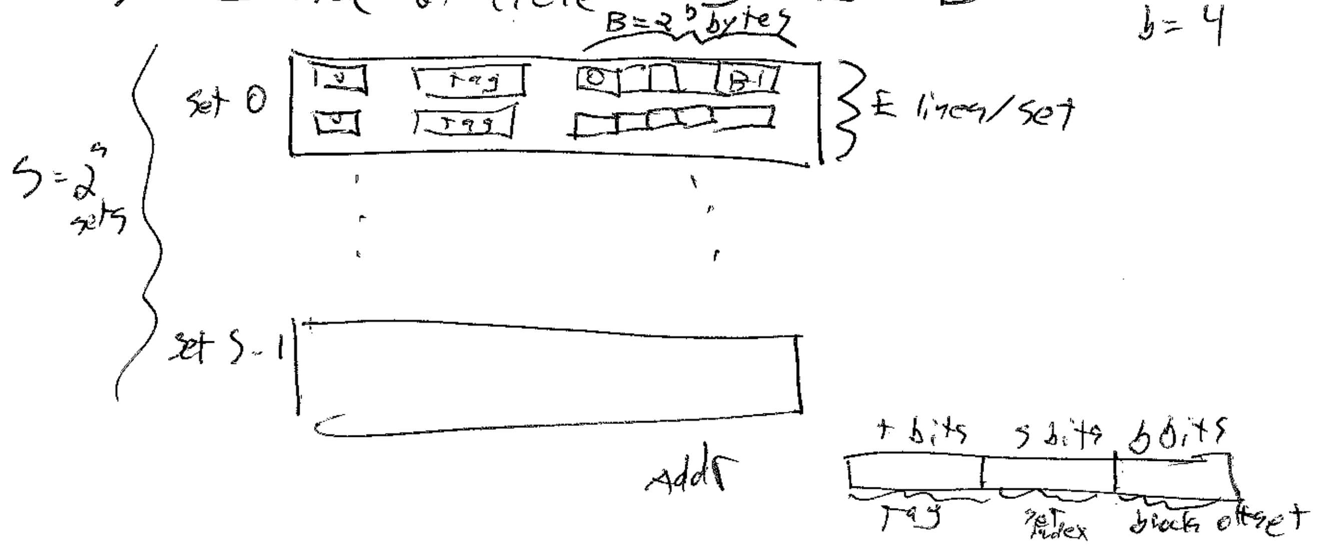
⇒ $M = 2^m$ unique addresses

- A cache organized to $S = 2^s$ cache sets
- Each set = E cache lines
- Each line = block of $B = 2^b$ bytes, a valid bit, & $t = m - (b + s)$ tag bits that uniquely identify the block stored in the cache line.

⇒ (S, E, B, m)

⇒ C = size of cache $= S \times E \times B$

$b = 4$



Direct-Mapped Caches

- $E = 1$ (1 line per set)
- conflict misses ⇒ thrashing!

```
for (i=0, j=0; i<N, j<M; i++, j++){
    Sum += a[i] * b[j]
```

⇒ a[0] → a[3] cached, then overwritten by b[0] → b[3]

⇒ overwritten by a[1] → a[4] ⇒ etc...

- This is especially important for arrays with size 2^x.
- To fix, add an empty buffer after array ⇒ the arrays map to different cash sets.

Set Associative:

- $E > 1$ ⇒ tag bit selects the line.
- Line replacement gets weird.

Fully Associative:

$S = 1$, $E = C/B$

- This is computationally expensive A F

Cache writing is more complicated:

- write through = updating 1 level updates all ~~the~~ lower ones.
  ⇒ huge bus w/ each write.
- write back = only write it to the next level when that line in the cache is replaced.
  ⇒ cache must have "dirty bit" that indicates if that line has been modified.

Textbook Notes 6.4 ctd

- what about write misses?
  "write-allocate" ⇒ load block from lower level, then ~~allocate~~ write
  "no-write-allocate" ⇒ go directly to lower level.
- write through caches = no write allocate
- write-back caches = write-alocate.
  ↑ assume this style when programming

- real world:
  · I-caches & D-caches = separate (and I = read only).

- Measuring Cache Performance:
  1. Miss rate (#misses/#references)
  2. Hit rate (1 − miss rate)
  3. Hit time (time to deliver a word in cache to CPU)
  4. Miss penalty (additional time after a miss)

  ⇒
- Trade offs:
  ↑ cache ⇒ ↑ hit rate, ↑ hit time
  ↑ Block size ⇒ ↑ hit rate for spatial locality, but less lines per set
  ⇒ ↓ hit rate for temporal locality.
  · Nodern caches = 32 - 64 bytes
  ⇒ ↑ miss penalty

- ↑ Associativity (↑E) ⇒ ↓ thrashing, ↑ cost, ↓ speed, ↑ miss penalty
- write through = simpler & use a write buffer ⇒ ↓ miss penalty
- write back ⇒ less transfers (more important at low levels)

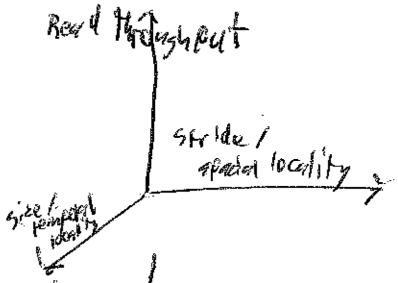
6.5 writing cache-friendly code

- Good locality ⇒ ↓ miss rate ⟺ faster
- Guidelines:
  1. Make the common case go fast.
  2. Minimize # of cache misses in each inner loop

- repeated references to local vars are good 'cause compiler can cache them in the reg. file (temporal locality)
- Stride-1 ref. patterns are good 'cause caches at all lvls of mem store data as contiguous blocks (spatial locality)

6.6 Putting it Together

- rate a prog. reads ~~mem~~ data from mem = read throughput or read bandwidth. (MB/s)
- Dig memory mountain code (p.622)
  - ↑ size → ↓ temporal locality
  - ↑ stride → ↓ spacial locality



- each computer's memory mountain is uniqe!
- eg core i7's
  - 4 ridges corresponding to regions of temporal locality where entire set fits in L1-3 cache & main mem.
  - full order of magnitude b/t highest L1 ridge & lowest main mem.
- try to exploit locality by getting your program to work in the peaks, not the valleys

Textbook Notes 6.6 ctd.

~~Class Notes 5/20~~

Rearranging Loops to ↑ spatial Locality

- Consider multiplying 2 n x n matrices

$$C = AB$$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\Rightarrow \quad c_{11} = a_{11} b_{11} + a_{12} b_{21}$$
$$c_{12} = a_{11} b_{12} + a_{12} b_{22}$$
$$c_{21} = a_{21} b_{11} + a_{22} b_{21}$$
$$c_{22} = a_{21} b_{12} + a_{22} b_{22}$$

⇒ 6 functionally equivalent ways to code it

eg ijk

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++){
        sum = 0.0;
        for(k=0; k<n ; k++)
            sum += A[i][k] * B[k][j];
        C[i][j] += sum;
    }
```

Results:

- miss rate = better predictor of performance than total # of memory accesses.
- Good locality ⇒ constant speed at high levels of n
  - cause intel programmers smart as shit
  - ⇒ mad stride-1 optimization

In general:

1. focus on inner loops (bulk of computation & memory accesses)
2. maximize spatial locality (read data objects sequentially w/ stride 1).
3. maximize temporal locality by using data object as often as possible after reading it from mem.

Textbook Notes 4.5 Pipelined Y86

- let's pipeline by adapting SEQ
  1. first change computation of the PC into the fetch stage

4.5.1 Rearranging the Computation stages (SEQ+)

- PC updates gotta come at the start, "SEQ+"
- make it compute the PC val for the <u>current</u> instruction.
- just circuit retiming

4.5.2 Inserting Pipeline Registers

- Adding registers b/t the stages

<u>1. F</u> holds predicted val of prog. counter

<u>2. D</u> b/t fetch & decode. holds info bout most recently fetched instr. for decode

<u>3. E</u> b/t decode & execute. Info bout recently decoded instr. for ~~exe~~ execute

<u>4. M</u> b/t execute & memory. Info bout recently executed instr. for mem stage & branch conditions & branch targets for conditional jumps.

<u>5. W</u> b/t mem & feed back paths supplying computed results to reg. file for writing & return addr. to the PC select logic w/ ret. instr.

4.5.3 Rearranging & Relabeling signals

- need multiple copies of signals like valC, srcA, & valE.
- M_stat = stat ~~val.~~ signal coming stored at M pipeline reg.
- m_stat = status signal generated by memory stages control logic block.

- note we merge into valA for pipeline reg. E by choosing either valP from reg. D <u>or</u> val read fro A port of reg. file.
  - reduces amount of state carried to reg.s E & M
- we also associate a status code w/ each instr. for exceptional events.

New PC Instructions:

- we want to get the next PC right after calculating the current one (would ⇒ throughput of 1 instr./cycle)
- But if fetched instr. is conditional branch we don't know whether or not to take it till after instr. taken execute stage.
- If instr == ret, we don't know where to go until instr. passes thru M sta

- Besides these,
  - for call & jmp, it's valC
  - for others it's valP.
  - we can predict whether condition jmp will be taken (PC ← valC or not (PC ← valP), but we need to deal with the case in which we're wrong
- This = branch prediction.
  - lots of wild systems
  - we'll assume conditionals are always taken... (PC ← valC

Textbook Notes Ch. 4.5 p.2

- this strat = "always taken" = 60% success
  "never taken" = 40%,
  "backwards taken, forward not-taken" (BTFNT) = 65% success
  (think about loops.
- w/ return, we just don't process any more instr.s until ret instr. passes through write-back stage.

4.5.5 Pipeline Hazards

- data hazards & control hazards

eg Prog 1	1	2	3	4	5	6	7	8	9	10
0x000 irmovl $10, %edx	F	D	E	M	W					
0x006 irmovl $3,%eax		F	D	E	M	W				
0x00C nop			F	D	E	M	W			
0x00d nop				F	D	E	M	W		
0x00e nop					F	D	E	M	W	
0x00f addl %edx					F	D	D	E M		W

when addl. decodes, %edx & %eax written

Prog 3

0x000 irmovl $10, %edx	F	D	E	M	W			
0x006 irmovl $3, %eax		F	D	E	M	W		
0x00C nop			F	D	E	M	W	
0x00d addl %edx,%eax				F	D	E	M	W

when addl. decodes, neither %edx or %eax written yet

⇒ wrong result.

Enumerating classes of data hazards:

1. prog. reg.s — cause reg. file read in 1 stage, written in another
2. Prog. counter — conflicts b/t updating & reading prog. counter ⇒ control hazards. ⇒ prediction ⇒ mispredicted branches / ret instruction "exception handling"
3. Memory — reads/writes occur in 1 stage ⇒ no conflict (unless self-modifying code...)
4. CC reg. — written in execut, ~~write~~ read in execute & mem ⇒ no hazards
5. Status reg — enables exception handling

⇒ overall, we only need to worry 'bout reg. data hazards, control hazards, & exception handling.

4.5.6 Stalling

- Holds back instr.s in the decode stage until instr ~~holds~~ generating its source operands have passed through write-back.
- stalling = holding back all subsequent instructions 'till others go through by adding a bubble
  - like a dynamically generated nop instr.

eg

		1	2	3	4	5	6	7	8	9	10	11	
0x000	irmovl $10, %edx	F	D	E	M	W							
0x006	irmovl $3, %eax		F	D	E	M	W						
	bubble			{	→	E	M	W					
	bubble					→	E	M	W				} bubbles
	bubble						→	E	M	W			
0x00c	addl %edx, %eax			F	D	D	D	D	E	M	W		
0x00e	halt				F	F	F	F	D	E	M	W	

- easy implementation, meh performance.

Text book Notes 4.5 p.3

4.5.7 Forwarding

- Instead until write is complete, it can pass the value about to be written to pipeline reg. E as the same operand.
- or even the val M_valE to pipeline reg. D
- this passing of result value directly from 1 pipeline stage to an earlier one is "data forwarding"

= avoids stalling, but additional data connections & control logic

- Can be used for values generated by ALU & destined for write port E.
  - Or for values read from mem. & destined for write port M.

- AKA from: (e_valE, m_valM, M_valE, W_valM, & W_valE) to: (valA, valB).
- see figs 4.49 to 4.51 for decode-state logic to determine which val to use (from. reg. file or a forwarded val?).

Text book Notes 4.5.8 Load / Use Data hazards

- One class of data hazards can't be handled just by forwarding 'cause mem reads occur late in pipeline.
- eg
  when mrmovl reads val ~~from~~ into %eax while next instr (eg addl) ~~use~~ needs this as a source operand
- This can be solved by combining stalling & forwarding
- This is called load interlocks
  - b/t forwarding & load interlocks, all data hazards are solved.

4.5.9 Exception Handling

- Excepting instruction = instr. causing the exception
- eg halt, invalid addr, & invalid instr.
- here, all instr.s prior to this should comple & none after should effect programmer-visible state.
- In bigger system, need invoke an exception handler

- In pipelined, it's tricky
  1. Can get exceptions trigged by multiple instructions at the same time.
     eg halt in fetch stage, invalid mem. addr. in memory stage.
     ⇒ which do we report? usually the one in the furthest stage

  2. Instr. is fetched, causes exception, but later is canceled due to mispredicted branch.

Textbook Notes 4.5 (p.4)

3 instr. following excepting instr. ~~instr~~ ~~follows~~ ~~alters~~ alters prog-visible state before excepting instr. completes.

- we can avoid all these problems by merging exception-handling logic into the pipeline structure
  ⇨ <u>Stat</u> code in each pipeline register.
- that exception status propagates through pipeline w/ the rest of the info for that instruction until the write-back stage.
- then the control logic stops execution
- upon discovery of exception in Mem or write-back, changes to CC reg. and data memory are disabled.
- This strategy solves all three problems

Textbook Notes 4.5.10

- Cycles per instruction (CPI).
  - Avg. # of clock cycles taken to do 1 instr.
  - Reciprocal of the avg. throughput (in picosecs)

$$= \frac{C_i + C_b}{C_i} = 1.0 + \frac{C_b}{C_i}$$

C_i = instr.s , C_b = bubbles, ⇒ $\frac{C_b}{C_i}$ = bubbles per instr.

$= 1.0 + lp + mp + rp$

lp = load penalty = avg. freq. w/which bubbles injected while stalling for load / use hazards

mp = mispredicted branch penalty = avg. freq. w/ which bubs injected when cancelling instr.s

rp = return penalty = avg. freq. w/which bubs injected for ret instr.

Cause	Name	Instr. freq.	Cond. freq.	Bubs	Product
Load/use	lp	.25	.20	1	0.05
mispredict	mp	.20	.40	2	0.16
return	rp	.02	1.00	3	0.06
Total penalty					0.27

⇒ CPI = 1.27

- Note mispredicted branches = more than half this penalty

Unfinished business

- Multi-cycle instr.s → e.g. 32 cycles for int. division
- Interacting w/ the mem system — self-modifying code.
  - we use virtual addressing.
  - we assume 1-cycle mem. access

Textbook Notes 8.1 exceptions

- Partly hardware, partly OS.
- exception = abrupt change in the control flow in response to some change in the processor's state.
- In I_{curr} (encoded in various bits) change in state = event
- Event egs:
  - virtual memory page fault
  - arithmetic overflow
  - div. by Ø.

  # assigned by processor designer

  - system timer goes off
  - I/O request completes.

  number assigned by OS designers
- event ⇨ indirect proc. call (exception) ⇨ exception table (jump table) ⇨ OS subroutine (exception handler)

⇨

1 of 3 things:

1. ret. to I_{curr}
2. ret. to I_{next}
3. handler aborts the program.

- starting addr. of exception table = in the exception table base register.
- when called, ret. addr. pushed onto stack. & processor state. for restoration. (if control → kernel, pushed on the kernel's stack).

Classes of Exceptions

1. Interrupt - signal from I/O device - Async - ret. to I_{next}
2. Trap - Intentional Exception - Sync - ret. to $\underline{I}_{next}$
3. Fault - Potentially recoverable err. - Sync - might ret. to I_{curr}
4. Abort - nonrecoverable err. - Sync - no ret.

Note: syscall function ⇒ do said function in <u>kernel</u> <u>mode</u> (impt if you're doing things requiring permissions).

<u>In Linux:</u> 256 exceptions

- 0 - 31 = Linux IA32 defined. 32 - 255 = OS
- 0 = div. err = fault. - 14 = page fault. - 11 = seg fault
- 128 = sys call (AKA 0x80)

- seg fault = ref. to undefined area in virtual mem. or tryin' to write to a read-only text seg.
- 1 = exit, 2 = fork, 3 = read, 4 = write, 5 = open, 6 = close, 7 = waitpid, 48 = execve, 20 = get pid, 37 = kill, 29 = pause (wait for sig), 27 = alarm
- Sys calls made ~~of~~ by app w/ a trap instr. called int n (index of kernel service)
  - see the table in usr/include/sys/syscall.h

- programs can use syscall function, but it's rare.
  - usually use a wrapper function
  - these = system-level functions

M

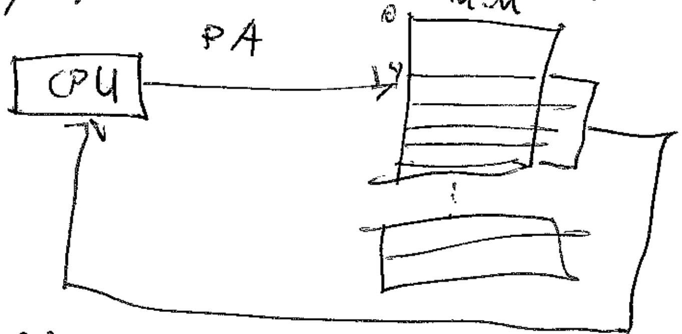
Textbook Notes 9.1

- modern processors use an abstraction of main mem. → virtual mem.
  ·Avoid mem. conflicts.
- ⇒ 3 important benefits:
  1. uses main memory efficiently (treating it as a cache for an addr. space stored on disks.
  2. Simplifies mem. management (providing each process w/ a uniform addr. space)
  3. Protects addr. space of each proc. from corruption by others.
- virtual mem. = central (all-pervasive)
- virt. mem. (VM) = powerful & dangerous

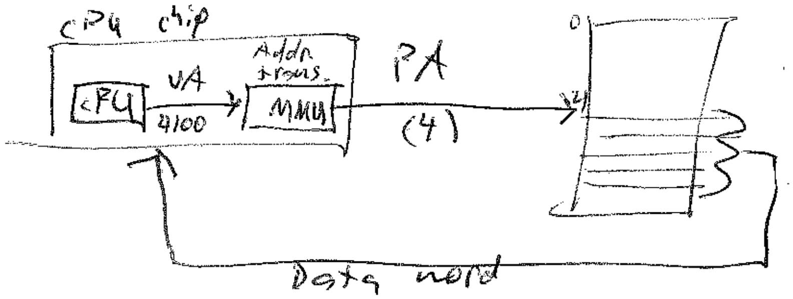
9.1 Physical & virtual addressing

- main mem. = array of M contiguous byte-sized cells.
- each has a physical addr. $(PA)_i \in [0 \div (M-1)]$

eg



- w/ VM, cpu generates a virtual addr. (VA), which is converted to the righ PA before being sent to M.M.
- The conversion = addr translation
- using a memory management unit (MMU) using a look-up table stored in main mem.
  ·contents of which are managed by the OS.



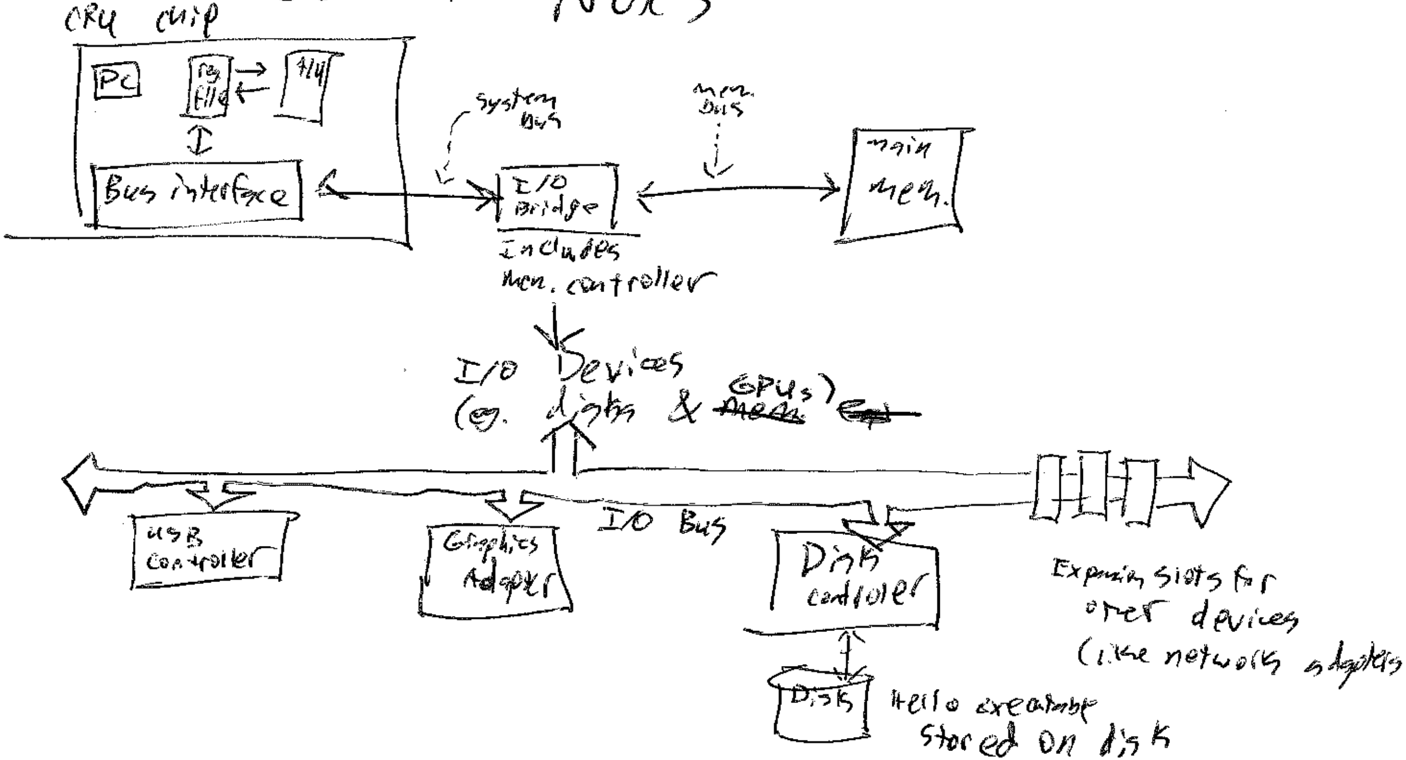
9.2 Addr. Spaces

- an addr. space = ordered ses of nonneg. int. addr.s {0,1,2,...}
- if consecutive, it's a linear addr. space.
- In VM, cpu generates virtual addr.s from addr. space of $N=2^n$ addr.s called "virtual addr. space". {0,1,2,...,N-1}
- size of an addr. characterized by # of bits needed to represent the largest addr. ⇒ n-bit addr space (usually 32-bit or 64-bit)
- Also a PA space. M-bytes {0,1,2,...,M-1}

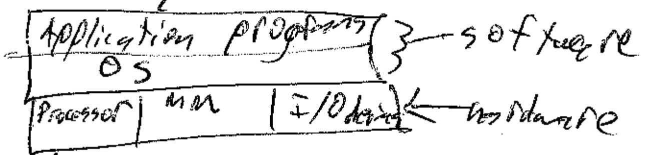
587 Non volatile memory

- DRAM & SRAM = volatile (lose info if supply voltage=off)
- Non-volatile mem.s retain info
  - ROMs
- PROM = 1 write (fuse w/ each mem. cell)
- EPROM = erasable PROM = can be rewritten on order of 1000 x
- EEPROM = electrically EPROM $\overset{\text{rewritten}}{=}$ order of 10^5 times
- flash mem. = fast, nonvolatile storage.
  - eg solid state disk (SSD)
- Programs stored in ROM = firmware (eg boot programs) & BIOS (Basic I/O system) routines

Text book Notes ch. 1



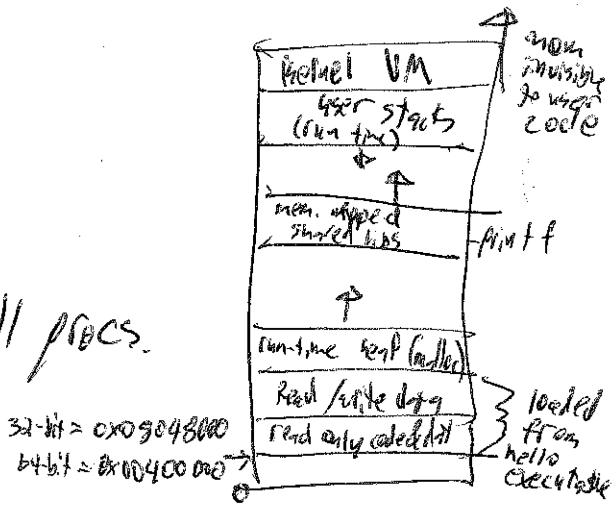
- CPU = central processing unit = processor
- OS = the layer of software b/t application program & the hardware.
  1. protectcts the hardware from misuse by runaway apps.
  2. provide apps w/ simple & uniform mechanisms for manipulating complicated & often wildly different low-level hardware devices



- files = abstractions for I/O devices
- VM = abstraction for both MM & disk I/O devices
- Processes = abstraction for processor, MM, & I/O devices

1.7.3 VM

- code begins at same location for all procs.
- followed by global vars (fixed memory)
- heap = dynamically allocated mem.



kernel VM
mem invisible to user code
user stacks (run time)
mem. mapped shared libs
printf
run-time heap (malloc)
Read /write data
Read only code&data
loaded from hello executable
32-bit = 0x08048000
64-bit = 0x00400000
0