# 8. VM (finished)

address: virtual L1 index, virtual L2 index, page offset

Page table Base reg: 指向 L1 Start 所在 PAddress

L1 $n^{th}$ entry: virtual $n^{th}$ L1 index 对应的 L2 table 的 physical page num

找到这一 page num 上的 L2 table 后:

L2 $m^{th}$ entry: virtual $m^{th}$ L2 index 对应的 page 的 physical page num

这一 physical page num 对应 page 上的 offset$^{th}$ entry 就是 PAddress of this address

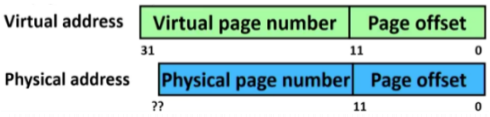
上面内容是这一 VAddress 对应的真实内容

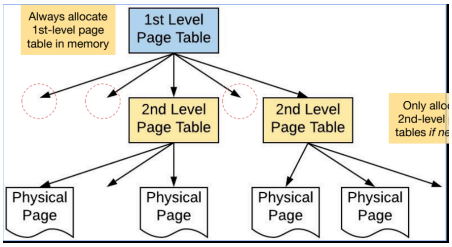
page offset: log_2(pageSize)

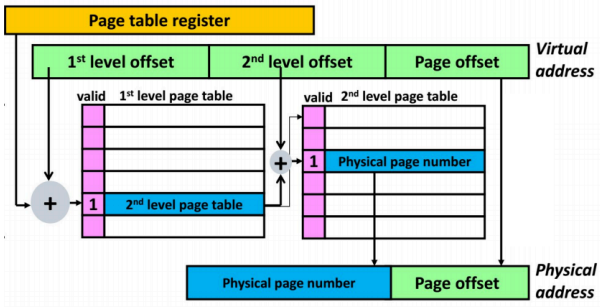
Virtual Page num bits: rest address

Physical Page num: physicalMemory / PageSize

Physical Page num index range: log_2(Physical Page num)




## Float num

mux: 0上1下

表示 float num:

1. most sig 0表示+, 1表示-;
2. exponential bits 表示 数字乘以 2 的多少次方. 偏移量 -127

   ex: 1000101 = 133, exponential = 133-127 = 6
3. 后面 Bits 表示 1.xxxx 的 xxxx

   ex: 6.67，6 = 110, 67= 101...

   0.67 × 2 = 1.34 → 1，余 0.34

   0.34 × 2 = 0.68 → 0，余 0.68

   0.68 × 2 = 1.36 → 1，余 0.36

   因而得到 110.101....., = 1.10101 * 2^2

   因而需要用 exponential bits 表示 129, 129-127=2

## 7.1 write back & allocation

overhead: non-data in a block / block size

**write back**: 对于 sw, 如果 cache 里有则写入 cache 并修改 dirty, LRU 时送回 memory； **allocate on write**: sw 即便 cache 里没有，也从 memory 里拉过来.

write through: sw 当作普通指令, 不通过 cache 而是直接 access memory.

write back, allocate on write 的劣势: overhead +1 dirty bit; 在 spatial, temporal locality 不好的 program 上 memory access 反而多于 write through

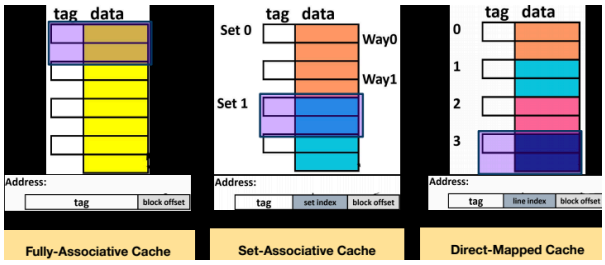
tag, set index, offset.
如果 inf length fully asso 仍 Miss: compulsory
Else: 如果 fully asso 仍 Miss: capacity； Else: Conflict
**Note: 更换 inf length 和 fully asso 时 set index 无, tag 长度改变.**

speed: flip flop(reg) > SRAM(cache) > DRAM(memory) > disk

计算 latency: aver latency = cache latency + mem latecy * missrate
temporal locality: 变量在同一段时间会多次使用;

spatial locality: 离得近的变量很可能接连被使用

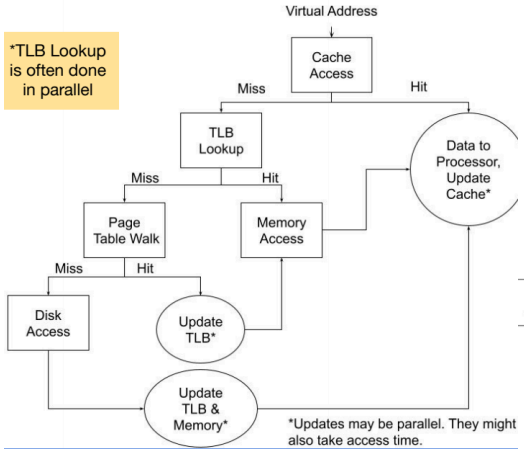
note: 总 cycles = #inst + #stages-1. ex:1000->1004

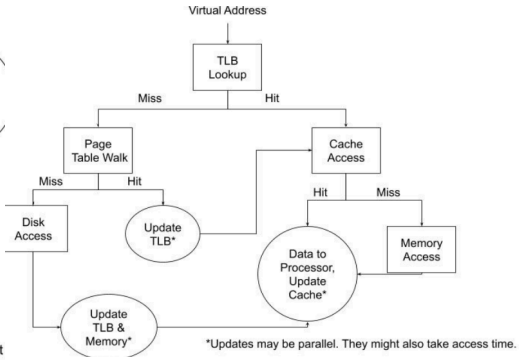



**Block Offset Bits** = $\log_2$(Block Size)
**Set Index Bits** = $\log_2$(#Sets)
**Tag Bits** = Address Size - (Set Index Bits + Block Offset Bits)
**Address Size** = $\log_2$(Size of Memory)
**Cache Size** = #Cache Blocks * Block Size
**#Cache Blocks** = #Cache Lines = #Sets * #BlocksPerSet
**#BlocksPerSet** = #Ways = Associativity
**LRU Bits** = $\log_2$(#BlocksPerSet)

**Page Offset Bits** = $\log_2$(Page Size)
**VPN Bits** = Virtual Address Size - Page Offset Bits
**PPN Bits** = Physical Address Size - Page Offset Bits
**Physical Address Size** = $\log_2$(Physical Memory Size)
**#Virtual Pages** = $2^{\text{VPN Bits}}$
**Physical Memory Size** = Physical Page Size * #Physical Pages
**#Physical Pages** = $2^{\text{PPN Bits}}$
**#Page Table Mappings** = #Virtual Pages

## Virtually addressed cache




**Size of Page Table** = #Entries * Size of an Entry

**ngle Level Page Table:**

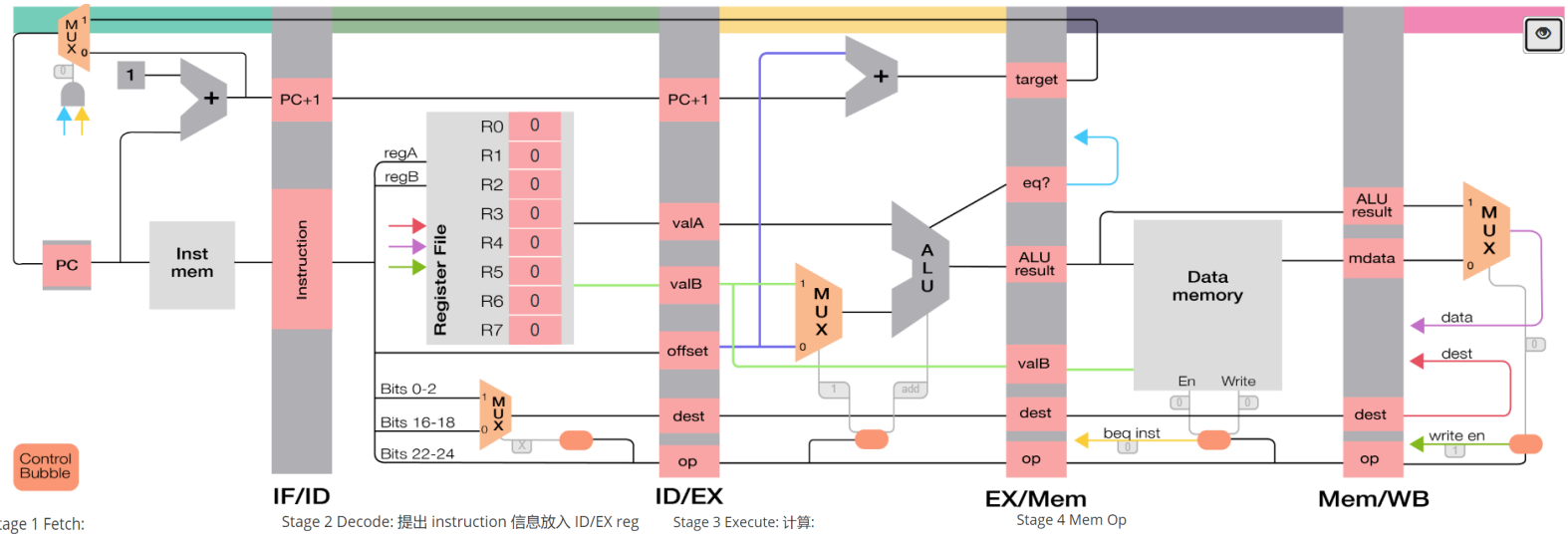
**#Entries** = #Mappings = #Virtual Pages
**Size of Page Table Entry** = Size of Control Bits + PPN bits

**ulti Level Page Table:**

**Total #Entries in Leaf Level** = #Mappings = #Virtual Pages
**Size of Leaf Level Page Table Entry** = Size of Control Bits + PPN bits
**Total #Entries in Intermediate Level** = #Page Tables in Next Level Down
**Size of Intermediate Level Page Table Entry** = Size of Control Bits + PPN bits/Physical Address Size

| Instr. | Read PC, Access Instr. Mem. | Read Reg. | ALU | Data Mem. Access | Write PC | Write Reg. |
|---|---|---|---|---|---|---|
| add | ✔ | ✔ | ✔ | | | ✔ |
| nor | ✔ | ✔ | ✔ | | | ✔ |
| lw | ✔ | ✔ | ✔ | ✔ (Read) | | ✔ |
| sw | ✔ | ✔ | ✔ | ✔ (Write) | | |
| beq | ✔ | ✔ | ✔x2 | | ✔ | |
| jalr | ✔ | ✔ | | | ✔ | ✔ |
| noop | ✔ | | | | | |
| halt | ✔ | | | | | |

add/nor/sw/beq: 4 cycles
lw: 5 cycles
noop/halt: 2 cycles




Stage 1 Fetch:

(1) index memory by PC address;
(2) PC++，把 PC和 instruction 写入 IF/ID

Stage 2 Decode: 提出 instruction 信息放入 ID/EX reg

(1) 根据 regA, regB 在 regfile 里提取 regA, B value;
(2) 提取 offsetfield (for beq,lw,sw);
(3) 提取 dest reg（16-18regB for lw, 0-2regC for add, nor)
(4) 提取 opcode, 继承PC

Stage 3 Execute: 计算:

(1) target = PC+offset ;
ALU = regA val + regB val/offset
(for beq) equal = (regA val == regB val)

(2) target, eq, ALU result 传入 Ex/Mem. valB(for sw). dest, op 继续继承入 Ex/Mem

Stage 4 Mem Op

(1) if beq equal, 传 target 入 IF/ID (即mux 选择target 而非 #1)

(2) sw: 直接把 valB 写入 ALU result address
for lw: 在 data mem 中取出 ALU result address 上的 data （memData)
把 ALU result. opcode, destReg(for lw, add, nor) 继承下来，放入 Mem/WB stage

Stage 5: WriteBack:
lw: 把 Mdata write back **to destReg**
add,nor: 把 ALU result write back **to destReg**
(remember that destReg is maintained)

Stage 2 Decode 从 regFile 读取 regs
Stage 3 Ex 对 regs, offset 进行运算
Stage 4 传 target，sw write into memory
Stage 5 Write back into Regfile(faster than Stage 2).

## 6.2 Data Hazard

Stall:

Decode 结束后, 发现前 1/2 Stage 上有 dependency 则留在 ID. dependency 在前1, 插入 2 noops.
Dependency 在前2，插入 1 noop. 等到 depency 在 Stage 5

Forward:

lw 需要 regA value；sw 需要 regB value；add, nor 需要 regA,B value；**需要位置都在 Stage 3.**

True Result 会被 add, nor, lw 的结果修改. add,nor true resule 存在 Stage 4 ALU result (Stage 5保留)；lw true result 存在 Stage 5 MData.

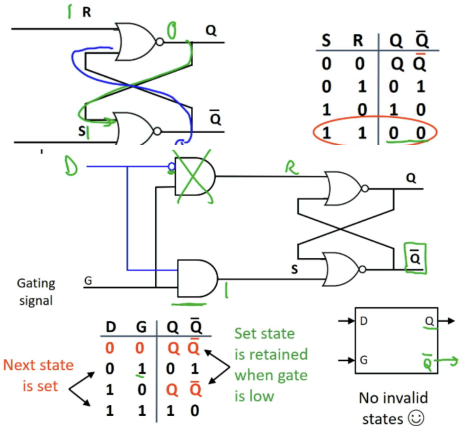
因而 add, nor, lw 从 Stage 4,5 导回 Stage 3；lw 从 Stage 5 导回 Stage 3. 如果 lw 后面紧跟一个 dependent，不得不让它 decode 时 stall 一回合.

## 6.3 Control Hazard

beq: Stage 1 Fetch 就需要信息, Stage 4 结束才获得
Stall 策略: beq 在 Stage 2,3,4, 分别向 Stage 1 插入一个 Noop
Squash 策略: 如果 Stage 3 结束查看 Eq 发现预测错误, 就在 beq stage 4 向 Stage 2,3,4 分别插入一个 Noop，它导回后重新执行正确 beq 后的指令. 一共多执行了 3 cycles.

同样的 program, 使用 detect and stall 和 code 中插入 noops 高有相同的 runtime，但是 CPI 比 code 中插入 noops 更高. 因为插入 noops 使得 instructions 数量变多；detect and forward runtime 和 CPI 都更低.

program with few branches benefit from ICache due to spatial locality; many loops benefit from ICache due to temporal locality.




Padding:

**Primitive object (int, char, etc):** 对于一个 size 为 N bytes 的 primitive object,

**存到下一个 mod N = 0 的 address 上就可以了。**

比如现在在 0x1001，下一个 object 是个 int， 就要跳到 0x1004 上

**对于 sequential object:** treat 每个元素 as independent object，一共只需要 padding 一次

Struct: 除了正常的 Padding 外，再保证

1. struct 的 starting address 是 struct 中的 largest primitive 的倍数
2. 整个 struct 的 total size 是 struct 中的 largest primitive 的倍数

`>>` 右移, 如果二补码那么最高位补上 0 还是 1 取决于最高位是 0 还是 1.

`sizeof(n)` 表示这个 n 的 **datatype 占的 bytes 数**（而不是 n 这个值本身占用的字节数）

not(A) = nor(A, A)

and(A, B) = not(or(not(A), not(B))) = nor(not(A), not(B)) = nor((nor(A, A)), (nor(B, B)))

or(A, B) = not(nor(A, B)) = nor(nor(A, B), nor(A, B))

LC2K:
Jalr 把 PC+1 存入 regB，branch to V[regA]; Halt: PC++

**overall delay 就是 delay 最大的一个串联路线的 delay.**

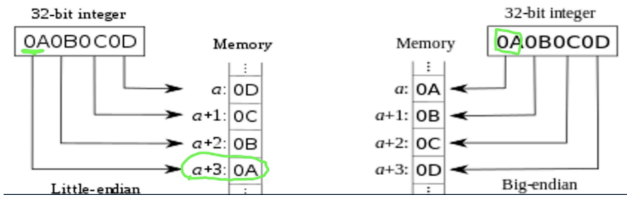
sizeROM = 2^{input size} * outputsize

Execution time = CPI * #insts * clockPeriod

## Big Endian && Small Endian

Endian 表示在一个 half/double/standard word 内，bytes 的 ordering: **significant bits 的地址在前面还是后面**

Little Endian 表示 word 的 4 个 bytes 中从前到后是 insignificant 到 significant；big endian 反过来




ARM 中两种都可以使用，只是要 consistent，我们默认使用 little

比如现在我们有两个 word 0xABCDEF12，0x12345678

那么:

| 0x1001 | 0x1002 | 0x1003 | 0x1004 | 0x1005 | 0x1006 | 0x1007 | 0x1008 | |
|---|---|---|---|---|---|---|---|---|
| 12 | EF | CD | AB | 78 | 56 | 34 | 12 | |

Symbol table: 列举了所有能够在这个这个文件外被看到的 labels. 比如 **function names, global variables 等.**

Relocation 需要做的：relocate absolute reference to reflect placement by the linker

(1) PC-relative 的 addressing (beq 等)：**never relocate.**

(2) Absolute Address (mov 等): always relocate

一个2's complement num 取负: nor 自身之后再 +1; 相对地, nor 自身的行为等价于取负后 -1.
两个数的 nor 在二进制表示位宽足够的情况下不受位宽影