

Computer Architecture II

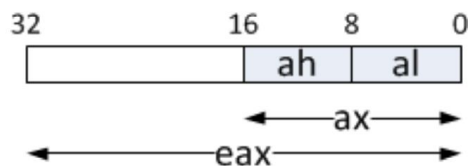
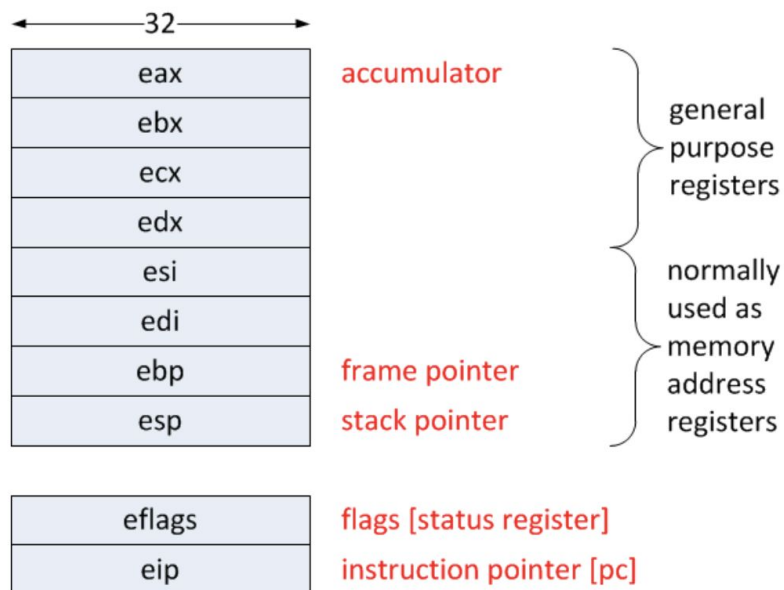
CS3021

INSTRUCTOR: Jeremy Jones
jones@scss.tcd.ie

IA32

- 32-bit CPU, performs 8, 16 and 32 bit integer arithmetic
- 32-bit address space 2^{32} bytes [4GB]

Registers



Instruction Format

add eax, ebx // $eax = eax + ebx$ [right to left]

Only the following operand arrangements are allowed:

- *register, register* -> *add eax, ebx*
- *register, immediate* -> *add eax, 2*
- *register, memory* -> *add eax, [ebp]*
- *memory, register* -> *add [ebp+8], ebx*

The following operand arrangements are **not** allowed:

- *memory, memory* -> *add [eax], [ebx]*
- *memory, immediate* -> *add [eax], 4*

Addressing Modes

Supported Addressing Modes

[a] = contents of
memory address a

| addressing mode | example | |
|-----------------|------------------------|-------------------------|
| immediate | mov eax, n | eax = n |
| register | mov eax, ebx | eax = ebx |
| direct/absolute | mov eax, [a] | eax = [a] |
| indexed | mov eax, [ebx] | eax = [ebx] |
| indexed | mov eax, [ebx+n] | eax = [ebx + n] |
| scaled indexed | mov eax, [ebx*s+n] | eax = [ebx*s + n] |
| scaled indexed | mov eax, [ebx+ecx] | eax = [ebx + ecx] |
| scaled indexed | mov eax, [ebx+ecx*s+n] | eax = [ebx + ecx*s + n] |

IA32 Examples

mov eax, [ebp+8] *// eax = [ebp+8]* *[contents of ebp+8 in memory]*

mov eax, 123 *// eax = 123*

mov [ebp+8], eax *// [ebp+8] = 123* *[store immediate to memory]*

*lea eax, [ebx+ecx*4+16]* *// eax = [ebx+ecx*4+16]* *[performs address calculation]*

xor eax, eax *// eax = 0* *[zero a register]*

test eax, eax *// AND eax with itself*

je <label> *// jump if zero*

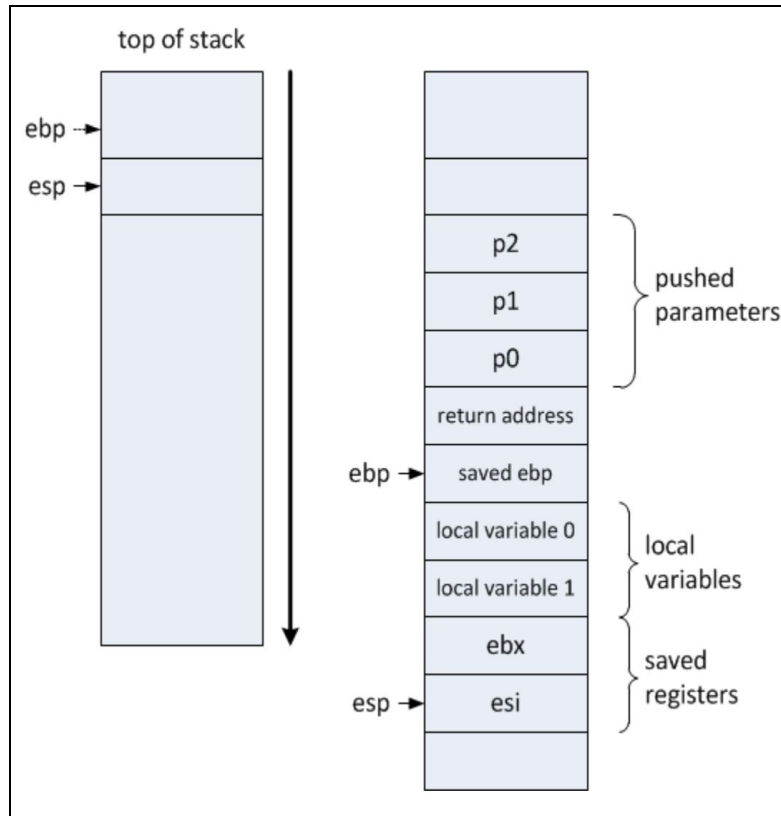
Function Calling

- Pass parameters *[Push onto stack]*
- Enter function *[Push ret address and jump to function]*
- Allocate space for local variables *[On stack by decrementing esp]*
- Save non-volatile registers *[Push onto stack]*

<FUNCTION BODY>

- Restore saved non-volatile registers *[Pop off stack]*
- De-allocate space for local variables *[Increment esp]*
- Return to calling function *[Pop return address]*
- Remove parameters *[Increment esp]*

Function Stack Frame



Parameters are pushed right to left onto the stack.

$p0 @ ebp + 8$

$p1 @ ebp + 12$

$p2 @ ebp + 16$

Local variables are accessed using a negative offset.

$local0 @ ebp - 4$

$local1 @ ebp - 8$

Function Example:

Consider the function $f(a, b, c)$:

```
push 3           // push values for a, b, c in right to left
push 2
push 1
call f           // call f(1, 2, 3)
add esp, 12      // when f returns update esp to remove parameters from stack
```

Inside of the function f would look like the following:

```
f:  push  ebp           // save ebp
     mov  ebp, esp      // ebp -> new stack frame
     sub  esp, 8         // allocate space for locals x, y
     push ebx           // save non-volatile ebx if used
```

```
     mov  eax, [ebp+8]   // eax = p0 (ebp+8)
     add  eax, [ebp+12]  // eax = p0 (ebp+8) + p1 (ebp+12)
     mov  [ebp-4], eax   // x = p0 (ebp+8) + p1 (ebp+12)
```

```
     mov  eax, [ebp-4]   // eax = x
     add  eax, [ebp-8]   // eax = x + y
```

N.B RESULT RETURNED IN EAX

```
     pop  ebx           // restore non-volatile ebx
     mov  esp, ebp      // restore esp
     pop  ebp           // restore previous ebp
     ret   0            // return from function
```

For more function examples see tutorial answers submitted throughout the year.

Pipelining

- Break each instruction into a series of small steps and execute in parallel
- Clock rate = time taken for longest step

5 Stage Pipeline

- **IF** - Instruction Fetch
- **ID** - Instruction Decode & Register Fetch
- **EX** - Execution & Effective Address Calculation
- **MA** - Memory Access
- **WB** - Write Back

| | | | | | | | | | |
|------------|----|----|----|----|----|----|----|----|----|
| <i>i</i> | IF | ID | EX | MA | WB | | | | |
| <i>i+1</i> | | IF | ID | EX | MA | WB | | | |
| <i>i+2</i> | | | ID | ID | EX | MA | WB | | |
| <i>i+3</i> | | | | IF | ID | EX | MA | WB | |
| <i>i+4</i> | | | | | IF | ID | EX | MA | WB |

The pipeline can stall however while data is read from memory if the memory access causes a cache miss (**hit**: 1 clock cycle, **miss**: 3 clock cycles)

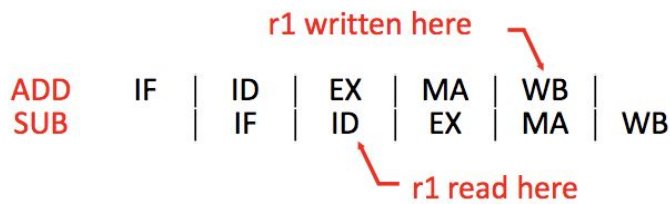
| | | | | | | | | | | |
|------------|----|----|----|-------|-------|----|----|----|----|----|
| <i>i</i> | IF | ID | EX | MA | MA | MA | WB | | | |
| <i>i+1</i> | | IF | ID | stall | stall | EX | MA | WB | | |
| <i>i+2</i> | | | ID | IF | IF | ID | EX | MA | WB | |
| <i>i+3</i> | | | | | | IF | ID | EX | MA | WB |
| <i>i+4</i> | | | | | | | IF | ID | EX | MA |

Data Hazards

Data hazards occur when instruction $x+1$ depends on a value used/changed in instruction x , for example:

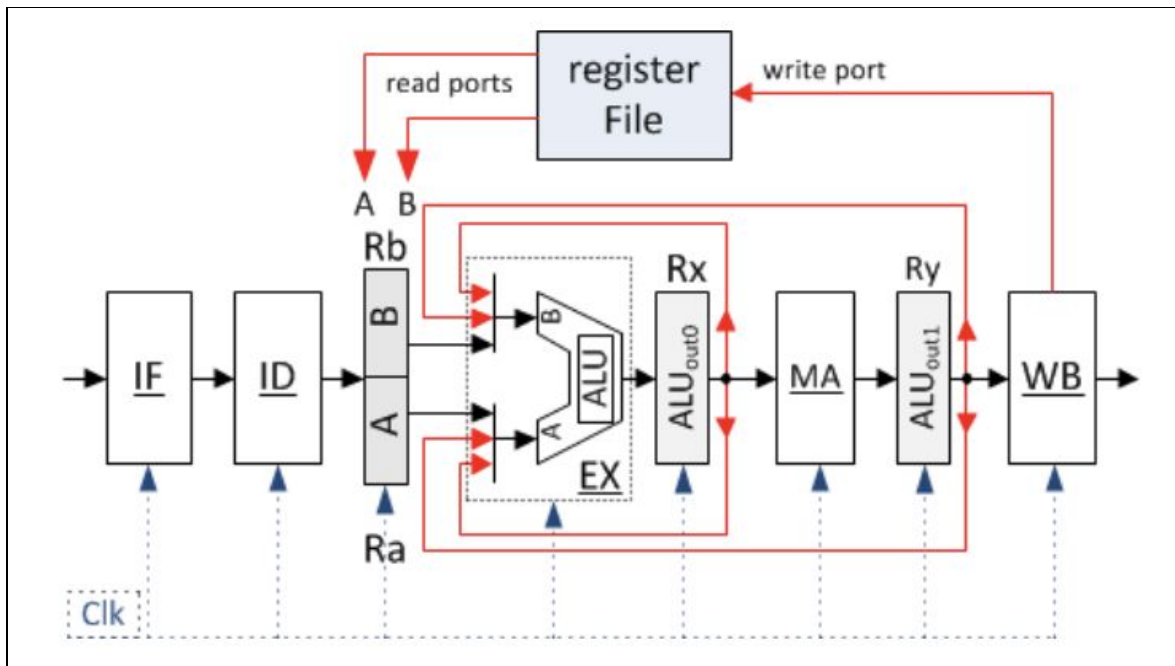
$r1 = r2 + r3$ **[ADD]**
 $r4 = r1 - r5$ **[SUB]**

The second instruction must wait for the first to complete and be written to memory before it can be safely used. If this is not accounted for the second instruction will be executed in the pipeline before the desired result is written to r1 from instruction 1.



This can be resolved by using a pipeline which forwards/bypasses writing to the register file and immediately passes the result of instruction 1 back to the ALU to be used in instruction 2.

Pipeline Forwarding



The ALU results from the “previous” two instructions can be forwarded to the ALU inputs as seen in the above diagram.

ALU_{out0} and ALU_{out1} are tagged as the destination register from each operation respectively.

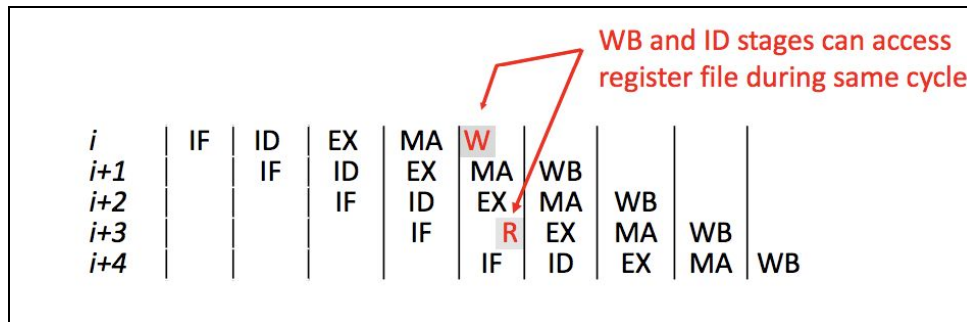
The EX stage of the pipeline checks for source register of the instruction in the order:

1. ALU_{out0}
2. ALU_{out1}
3. A/B

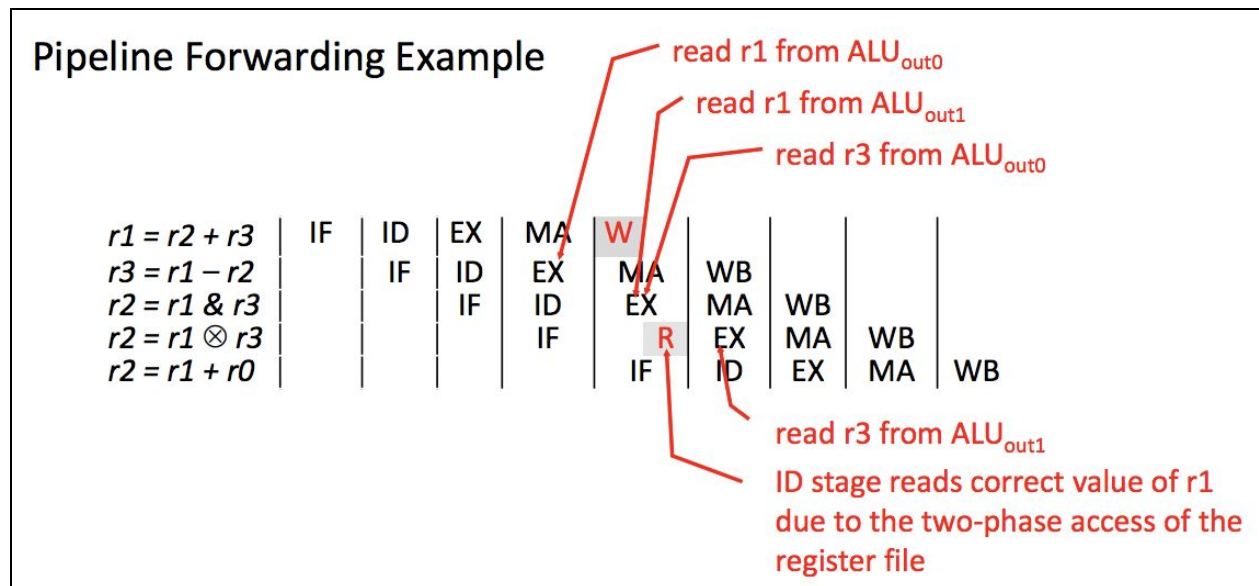
Two Phase Clocking

The DLX register file can be written and then read in a single clock cycle as follows:

- Written during the first ½ of cycle (WB phase)
- Read during the second ½ of cycle (ID phase)



Pipeline Forwarding Example




The first instruction writes to *r1* and the next 4 instructions use *r1* as a source operand and as a result access this via ALU_{out0} and ALU_{out1}.

Second instruction writes to *r3* which is used as a source operand by the third and fourth instructions and as a result are also accessed via ALU_{out0} and ALU_{out1}.

Load Hazards

When values are loaded from memory into a given register and are then used in the next instruction it produces a load hazard as the processor must stall for 1 cycle while the value is loaded into the register. Consider the following instructions:

| | | |
|---|-------------------------------|--------------------------|
|  | <code>r1 = M[a]</code> | <code>// load</code> |
| | <code>r4 = r1 + r7</code> | <code>// add</code> |
| | <code>r5 = r1 - r8</code> | <code>// subtract</code> |
| | <code>r6 = r2 & r7</code> | <code>// and</code> |

Here r1 is used on instruction 2 and as a result instruction 2 is dependent on the resulting value in r1 and must wait for it to be loaded.

| | | | | | | | | | |
|-------------------------|----|----|----|-------|----|----|----|----|----|
| <i>r1 = M[a]</i> | IF | ID | EX | MA | WB | | | | |
| <i>r4 = r1 + r7</i> | | IF | ID | stall | EX | MA | WB | | |
| <i>r5 = r1 - r8</i> | | | IF | IF | ID | EX | MA | WB | |
| <i>r6 = r2 & r7</i> | | | | | IF | ID | EX | MA | WB |

To avoid this it is often possible to reschedule the instruction causing the delay. For example with the above program instruction 4 can be swapped with instruction 2 and then there will be no data dependency issues and thus no stall.

DLX Pipeline Operation

- ALU instructions

| | |
|----|--|
| IF | $IR \leftarrow M[PC]; PC \leftarrow PC+4$ |
| ID | $A \leftarrow R_{SRC1}; B \leftarrow R_{SRC2}; PC1 \leftarrow PC; IR1 \leftarrow IR$ |
| EX | $ALU_{OUT0} \leftarrow \text{result of ALU operation}$ |
| MA | $ALU_{OUT1} \leftarrow ALU_{OUT0}$ |
| WB | $R_{DST} \leftarrow ALU_{OUT1}$ |

- Load/Store instructions

| | |
|----|---|
| IF | $IR \leftarrow M[PC]; PC \leftarrow PC+4$ |
| ID | $A \leftarrow R_{SRC1}; B \leftarrow R_{DST}; PC1 \leftarrow PC; IR1 \leftarrow IR$ |
| EX | $MAR \leftarrow \text{effective address}; SMDR \leftarrow B$ |
| MA | $LMDR \leftarrow M[MAR] \text{ or } M[MAR] \leftarrow SMDR$ |
| WB | $R_{DST} \leftarrow LMDR$ |

- BNEZ/BEQZ instructions [conditional branch]

| | |
|----|--|
| IF | $IR \leftarrow M[PC]; PC \leftarrow PC+4$ |
| ID | $A \leftarrow R_{SRC1}; B \leftarrow R_{SRC2}; PC1 \leftarrow PC; IR1 \leftarrow IR$ |
| EX | $ALU_{OUT0} \leftarrow PC1 + \text{offset}; \text{cond} \leftarrow R_{SRC1} \text{ op } 0$ |
| MA | if (cond) $PC \leftarrow ALU_{OUT0}$ |
| WB | idle |

DLX Branches

A simple DLX branch instruction results in a 3 cycle stall per branch instruction as the branch requires the new effective address to be calculated and the processor to adjust to this address.

| <i>branch</i> | IF | ID | EX | MA | WB | | | | | | | |
|---------------|----|----|-------|-------|----|----|----|----|----|----|----|--|
| <i>i1</i> | | IF | stall | stall | IF | ID | EX | MA | WB | | | |
| <i>i2</i> | | | | | | IF | ID | EX | MA | WB | | |
| <i>i3</i> | | | | | | | IF | ID | EX | MA | WB | |

The new PC is not yet known until the end of MA when it is read from memory. Once this has completed the program can continue executing.

To reduce this delay we need to determine if branch is taken or not taken earlier in pipeline and calculate the effective address during the pipeline.

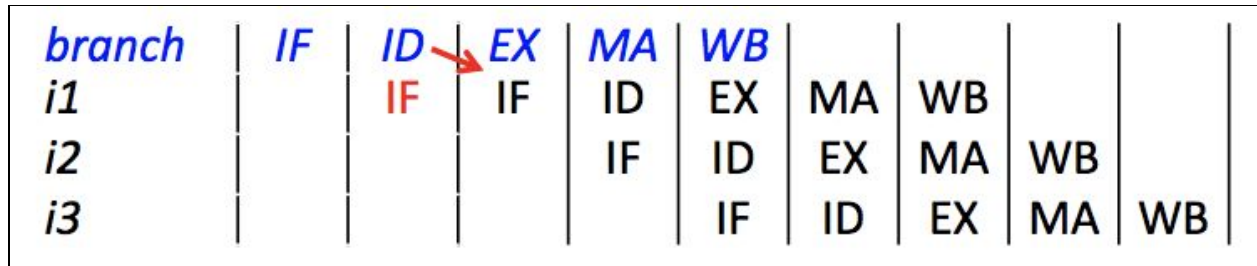
To perform a branch in a DLX processor we must first perform a “set conditional” instruction followed by a BEQZ or BNEZ instruction. This can be seen below:

```
SLT      r1, r2, r3 ; r1 = (r2 < r3) ? 1 : 0
BEQZ     r1, L      ; branch to L if (r1 == 0)
```

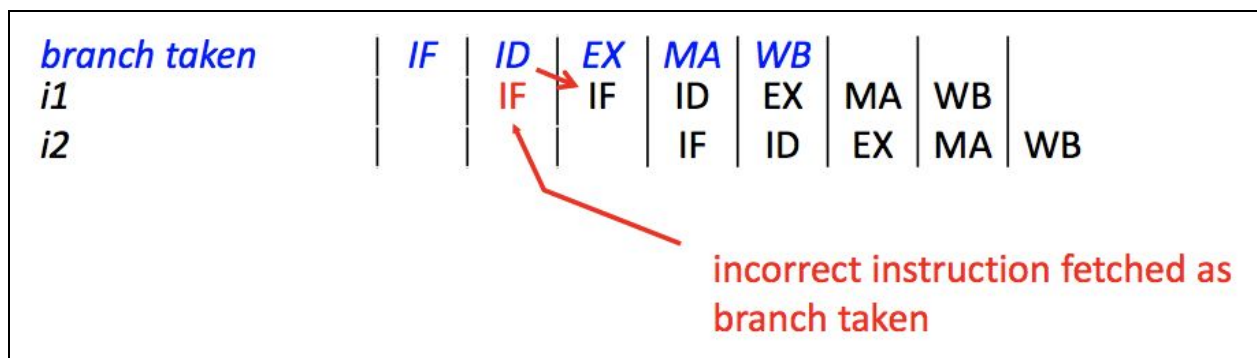
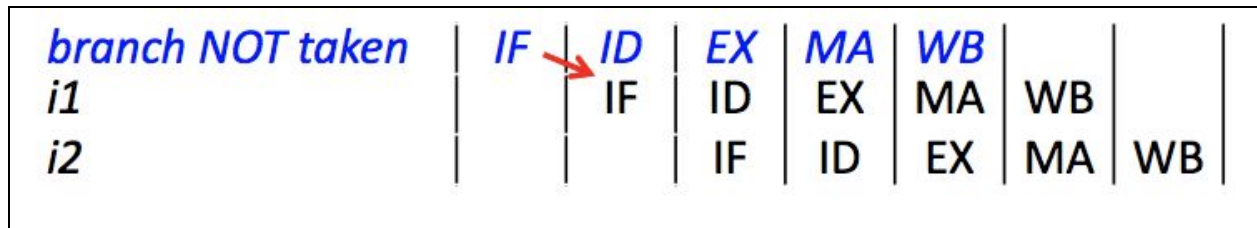
To implement this branch prediction you can add additional hardware to resolve branches during the ID stage of the pipeline. We must check if a register equals / !equal to 0 and adds an offset to the PC if the condition is true:

| | |
|----|--|
| IF | $IR \leftarrow M[PC]; PC \leftarrow PC+4$ |
| ID | if ($R_{SRC1} == / != 0$) $PC \leftarrow PC + offset$ |
| EX | idle |
| MA | idle |
| WB | idle |

This new addition now results in only a one cycle branch penalty as can be seen from the pipeline below. This stalls the pipeline 1 cycle until the branch target is calculated and known.



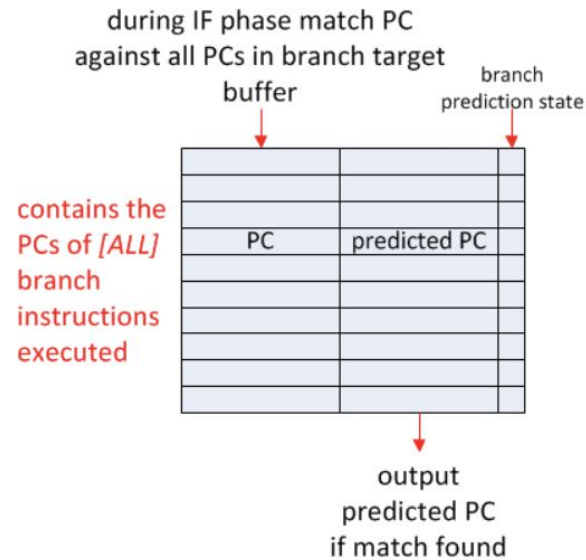
This can be further improved by assuming the branch is NOT taken. This way the pipeline only stalls if the branch is taken. We must also be careful to undo any side effects if branch is taken.



Here it can be seen that the pipeline only stalls in the case that the branch is taken. Otherwise it does not stall and continues as normal.

Branch Prediction

To implement branch prediction we must use a branch target buffer which can be used to resolve the branch location during the IF phase.

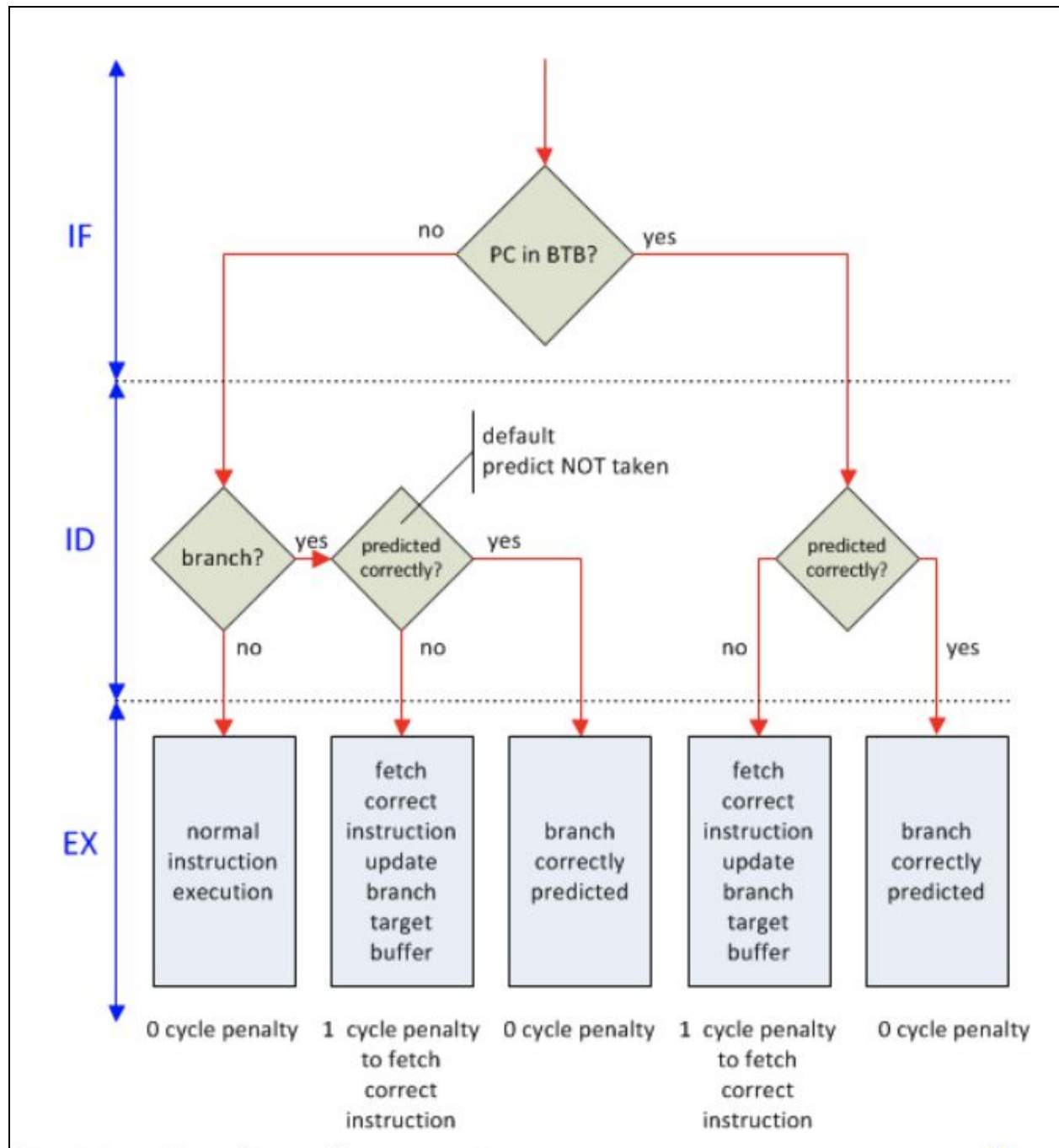


During an IF fetch, look for PC in branch target buffer. If match found, use predicted PC to fetch next instruction (take a guess).

If branch correctly predicted, NO pipeline stall.

If branch incorrectly predicted, must abort fetched instruction and fetch correct one (pipeline stalled for one clock cycle).

The branch target buffer must be updated if a *new branch* is fetched or a prediction changes.



Branch Prediction Example

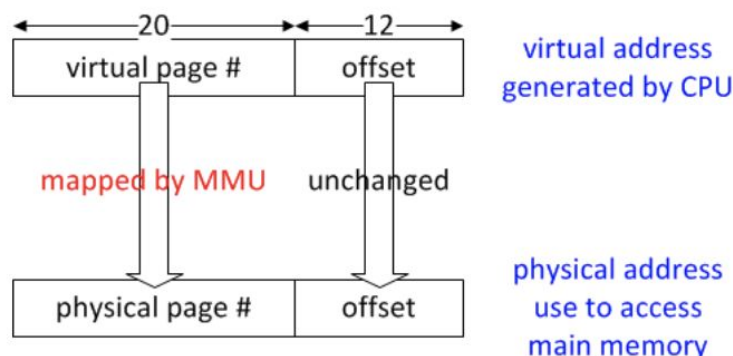
Consider the following loop:

| | | |
|-----|------|-------------|
| | add | r0, #10, r1 |
| L1: | ... | |
| | ... | |
| | sub | r1, #1, r1 |
| | bnez | r1, L1 |

- Assume BTB empty, predict a branch will branch same as last time
- First BNEZ execution, predicted incorrectly (default is branch NOT taken)
- Next 8 times BNEZ predicted correctly (predict taken, branch taken)
- Next time BNEZ predicted incorrectly (predict taken, branch NOT taken)

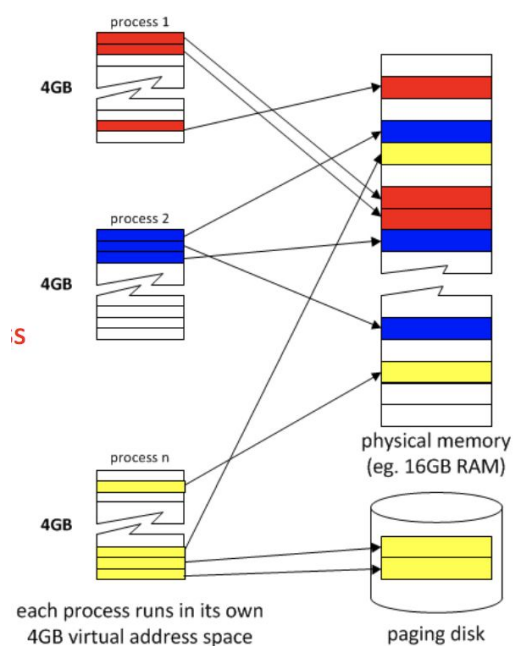
Memory Management Units (MMUs)

Memory Management Units converts a virtual address generated by the CPU into a physical address which maps to the memory system.



Address space is divided into fixed sized pages. The low order of an address represents an offset within a page. This is not affected by the MMU.

Each CPU core will typically have separate MMUs for instruction and data accesses. Virtual and physical address spaces do not have to be the same size.



Each process runs in its own 4GB virtual address space which has pages mapped by the MMU onto real physical pages in memory. Pages are allocated and mapped on demand by the Operating System.

Virtual pages in a process may be:

- Not allocated/mapped
- Allocated in physical memory
- Allocated on paging disk

Most processes use significantly less memory than the 4GB that is allocated.

An operating system keeps the “*working set*” of a process in physical memory to [minimise the page-fault rate](#).

Physical memory can be viewed as a cache to the paging disk.

Every page used in a process’ virtual address space must be mapped to an equivalent page in physical memory or on a paging disk.

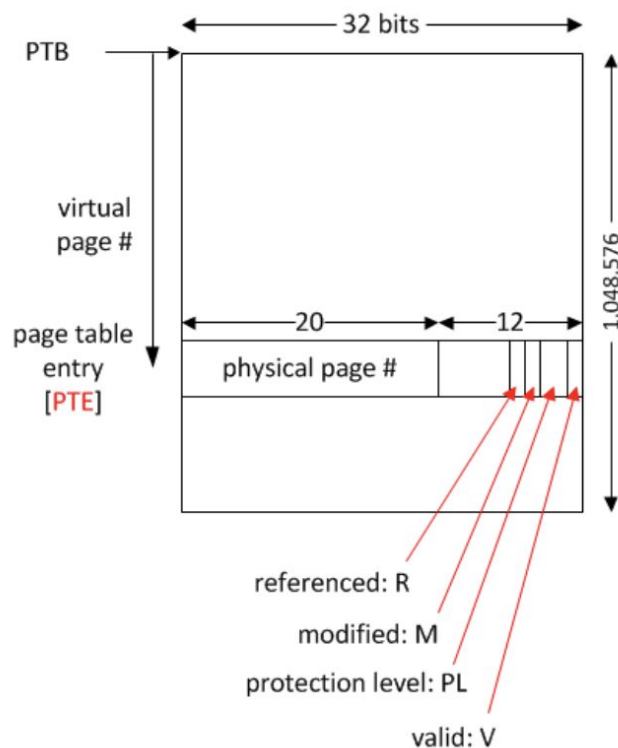
Generic MMU Operation (IA32, x64, MIPS...)

Virtual page numbers are converted to physical page numbers by table look-up.

Virtual page numbers used as an index into a page table stored in physical memory.

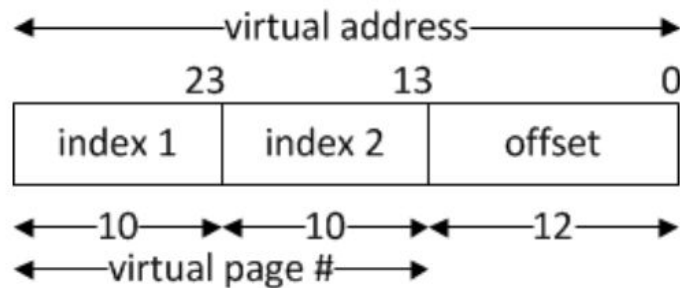
There is a page table for each individual process.

[Page table base register](#) PTB contains the physical address of the page table of the current running process. 4MB physical memory needed to hold the page table of every process.

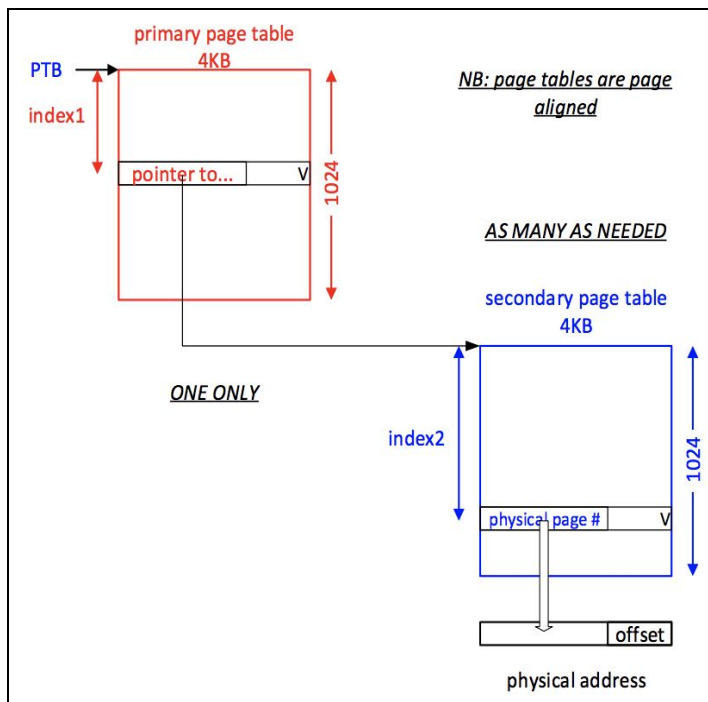


N-Level Page Table

To reduce the size of the page table structure an n-level look up table is used. This means that the larger the process, the more memory is needed for its page tables. However, the smaller the process, the less memory needed.



- Index 1: Primary page table
- Index 2: Secondary page table
- [Index 3-n]: Etc



PTB now points to the primary page table of the current running process. A valid primary page table entry (PTE) now points to a secondary page table.

Each process has one primary page table + multiple secondary page tables. Secondary page tables are created on demand.

When an MMU accesses a PTE it checks the **V**alid bit

If `is_primary_pte(PTE) && V == 0` (i.e PTE is Invalid)

- then no physical memory allocated for corresponding secondary page table

If `is_secondary_pte(PTE) && V == 0`

- then no physical memory allocated for referenced page (virtual address not mapped to physical memory)

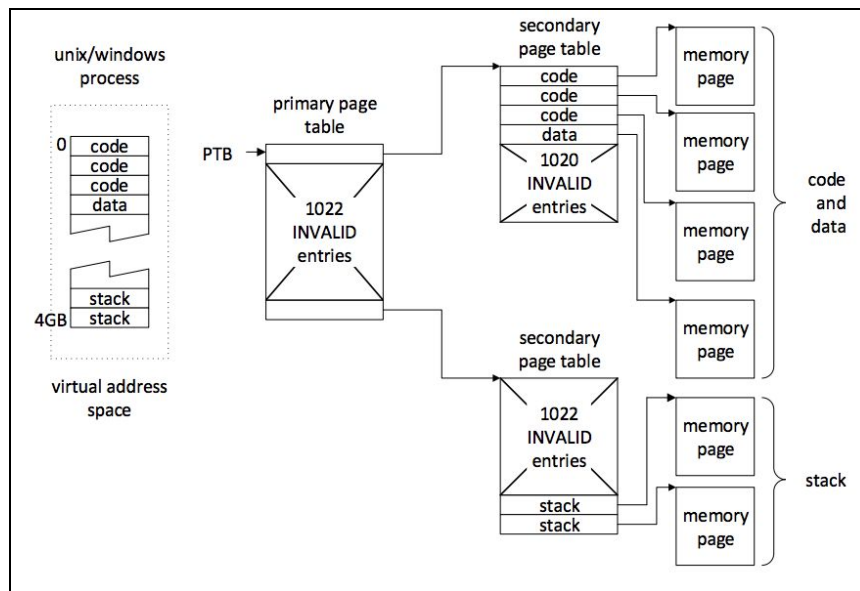
In both of the above cases a **page fault** occurs.

Page Fault Handling

An OS must resolve a page fault by performing one or more of the following actions:

- Allocate page of physical memory for use as a secondary page table
- Allocate a page of physical memory for the referenced page
- Update the associated page table entry/entries
- Read code/data from disk to initialise the page contents
- Signal access violation
- Restart or continue the fault instruction

Process Page Table Structure

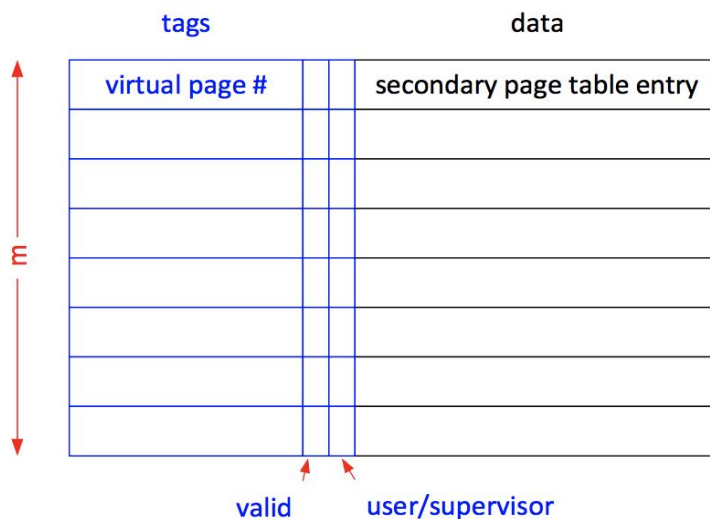


Each small process needs the following:

- 3 x Code Pages
- 1 x Data Page
- 2 x Stack Pages

This requires 2 secondary page tables to map code and stack areas.

Translation Look Aside Buffer (TLB)



Without an internal TLB each virtual->physical translation requires 1 memory access for each level of page table.

MMU contains an on-chip translation cache (TLB) which provides direct mappings for the m most recently addressed virtual pages.

All comparisons are performed in parallel. This allows for instantaneous address translations. If a match is not found page tables are walked by the CPU/MMU and the least recently used (LRU) is replaced.

RISC TLB Miss Handling

When a miss occurs most RISC's will generate an interrupt and perform a page table walk. In such cases page table structures are more flexible and can be set by software as it is not hard-wired into the CPU (could use a hash table).

A typical 64 entry fully associative TLB has a hit rate of > 90%.

TLB Coherency OS Implications

When a process is switched the TLB must be invalidated. This is as a result of all processes using the same virtual addresses.

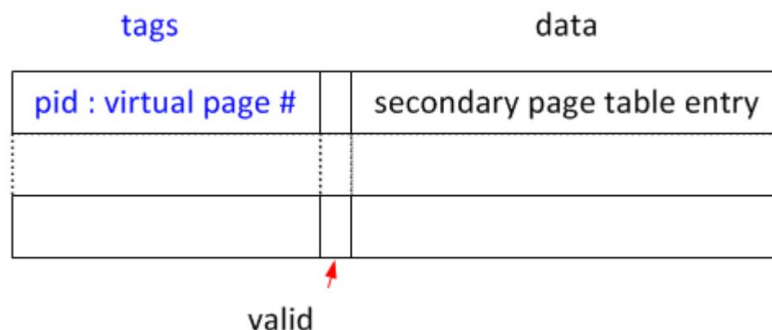
E.g Process 0 virtual address 0x1000 is NOT mapped to the same physical memory location as Process X virtual address 0x1000.

Whenever the page table base register (PTB) changes ALL corresponding TLB entries are invalidated.

If a page table entry is changed in main memory the OS must ensure this is reflected in the TLB.

Multiple Process Sharing TLB

It is possible for multiple processes to share TLB if the process ID (PID) is appended to the virtual page # as part of the tag.



Referenced and Modified Bits

CPU/MMU automatically determines the PTE **R**eferenced and **M**odified bits.

PTE changes are '*written through*' to corresponding PTE in physical memory.

CPU/MMU never clears the Referenced/Modified bits, this is up to the OS.

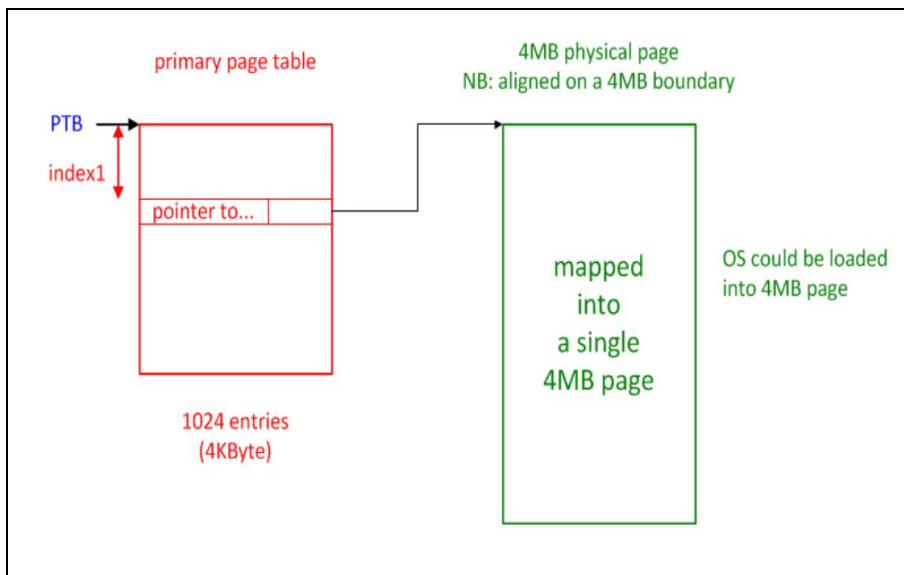
OS uses R/M bits to determine:

- Which pages are good candidates to be overwritten (least recently referenced)
- If pages need to be written to paging disk (no need to save if a read only code page)

Support for Different Page Sizes

Useful if MMU supports a number of different page sizes. Large pages allow a single TLB entry to map a large virtual page onto similar sized area of memory.

IA32 Solution:



- First level PTE points to a 4MB page of physical memory (not a second level page table)
- Bit set in primary PTE to indicate that it points to a large table (not a second level page table)

Breakpoint Registers

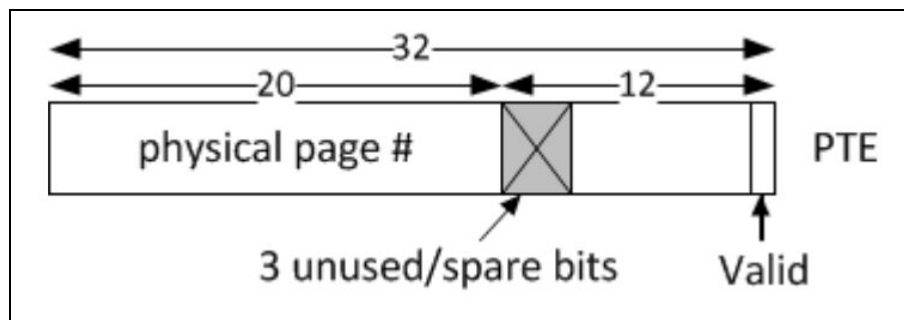
The MMU supports *breakpoint address registers* and *breakpoint control registers*.

The MMU can generate an interrupt if the breakpoint address is read or written or executed.

These are used for debugging purposes in real-time with breakpoints and watchpoints.

Integrating MMU and OS

The MMU's PTE's usually have a number of bits set aside for use by the OS implementer.



These spare bits can be used to define OS specific PTE types. Consider 2 spare bits V where V can == 0 or == 1.

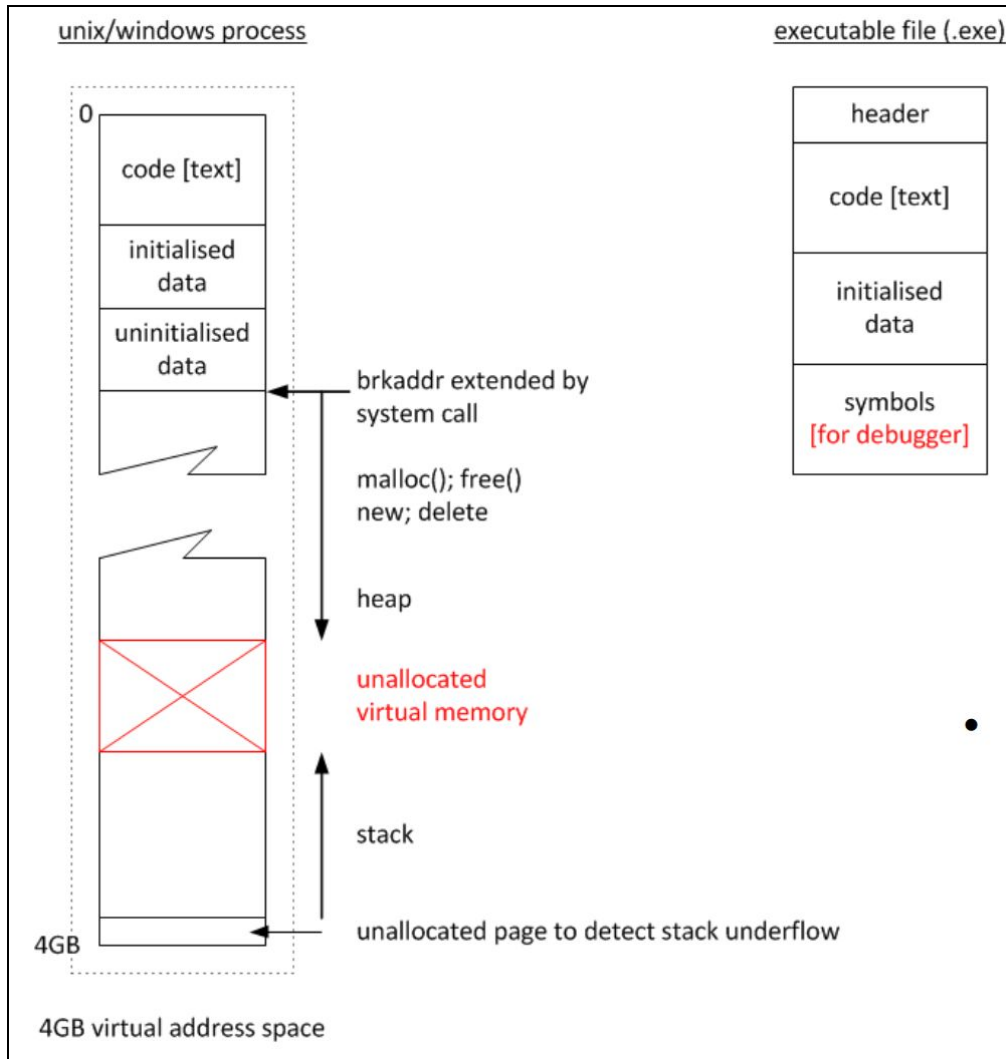
Types when V==1 [VALID]

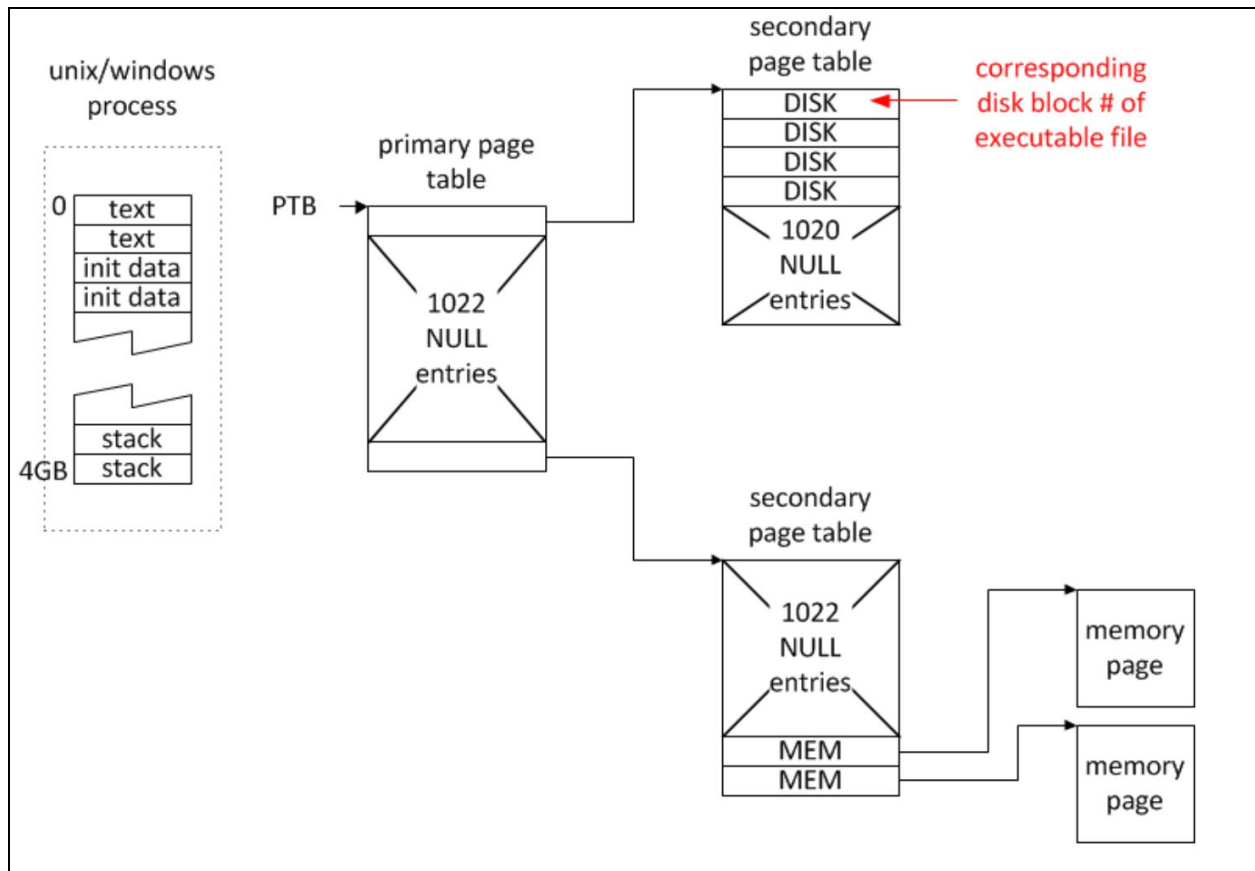
- MEM - Maps virtual to physical address
- LOCK - Same as MEM except page is locked (in use somewhere else)
- SPY - Maps virtual address to specific physical address (device drivers etc)

Types when V==0 [INVALID]

- NULL - Page not yet mapped to physical memory
- DISK - Not yet in memory, stored on paging disk
- IOP - Disk I/O is in progress
- SPT - Shared PTE

Initial Mapping of Unix Process





- **Text and init** data PTE's initialised to **DISK**
- **Stack** data (variables, args) initialised to **MEM**
- **Remaining** PTE's initialised to **NULL**

This process was allocated 5 pages of physical memory initially:

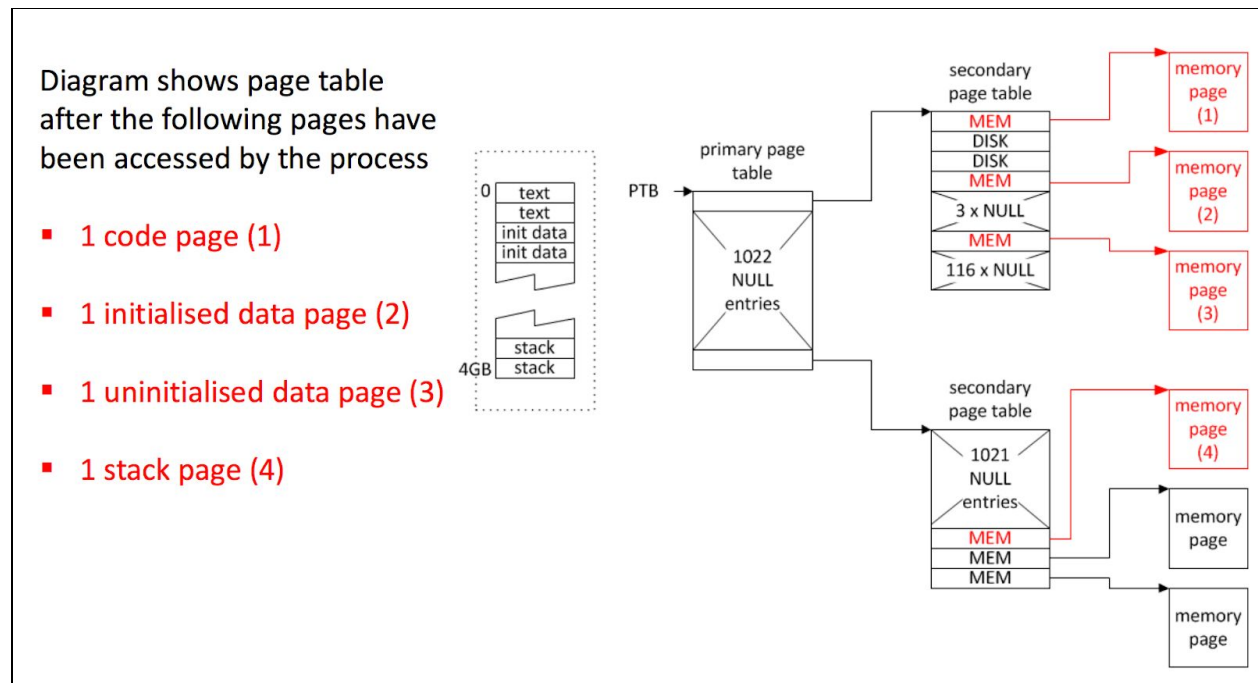
- 1x Primary page table
- 2x Secondary page tables
- 2x Stack Pages

Further tables are allocated to the process on demand.

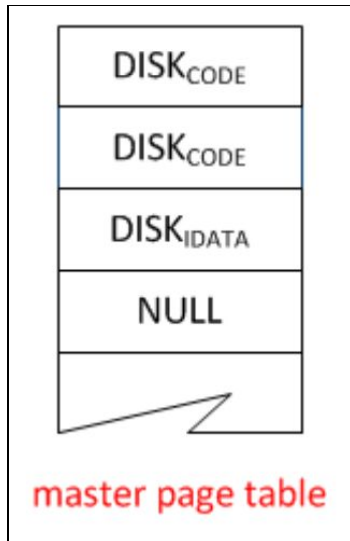
Initial Execution of Unix Process

After the process begins execution it will instantly generate a page fault as the page containing first instruction is still on disk. Page faults will continue to occur and each PTE type fault will be handled as follows:

- **DISK**: Allocate a page of physical memory and fill with data read from disk. DISK → IOP → MEM
- **NULL**: Physical memory has yet to be allocated. Checks if virtual address is in uninitialised data, heap or stack. If not considered illegal access violation. Otherwise a page of zeroed physical memory is allocated by OS.
- **MEM**: Protection level fault (writing to text via a NULL pointer)
- **IOP**: Wait for I/O to complete
- **SPY/LOCK**: Protection level fault



Text/Code Sharing



If the same process is executing more than once, ONLY a single shared copy of the code is needed in memory.

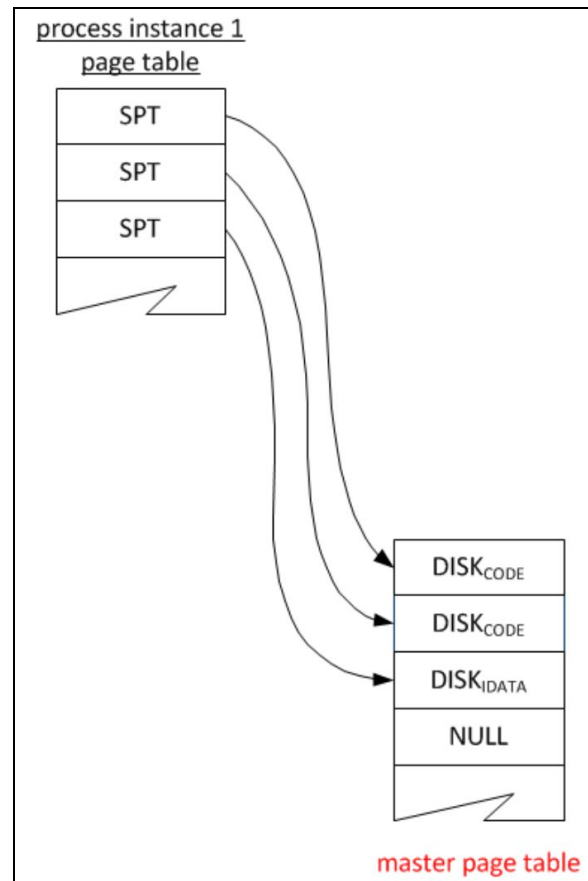
Each process still needs its own page tables for its data, heap and stack. Initialised data can be shared if it is read-only.

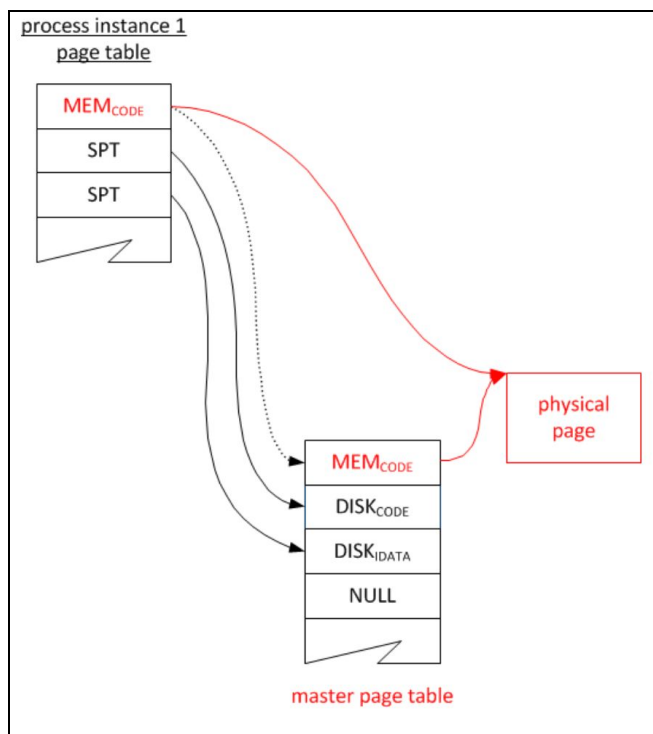
When a process is first executed a master page table is created. The PTE's for code and init data are assigned type DISK, remaining PTE's NULL.

A process page table is created by initialising its code and init data PTE's to type SPT (pointer). They point to their corresponding entries in the master page table.

Physical pages for the initial stack are attached to the process page table and the remaining PTE's are set to type NULL.

On a SPT page fault, the OS follows the SPT entry to the corresponding PTE in the master page table. The action performed depends on the entry type.

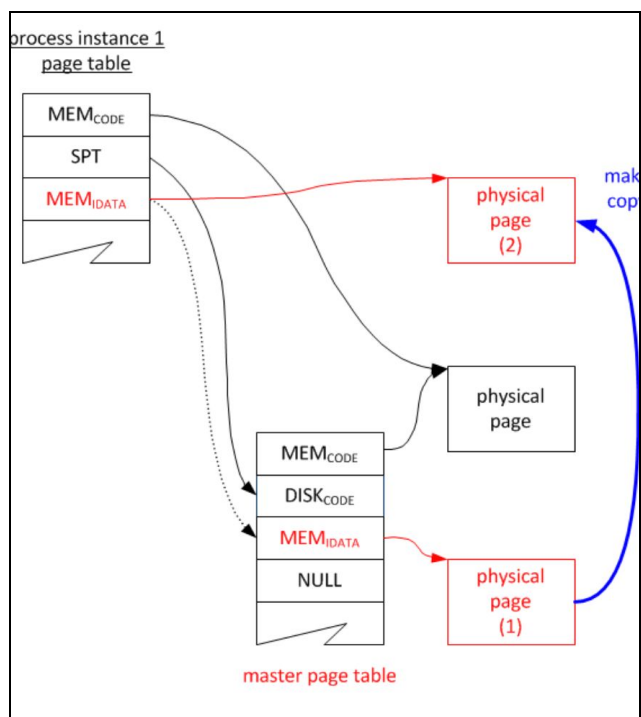


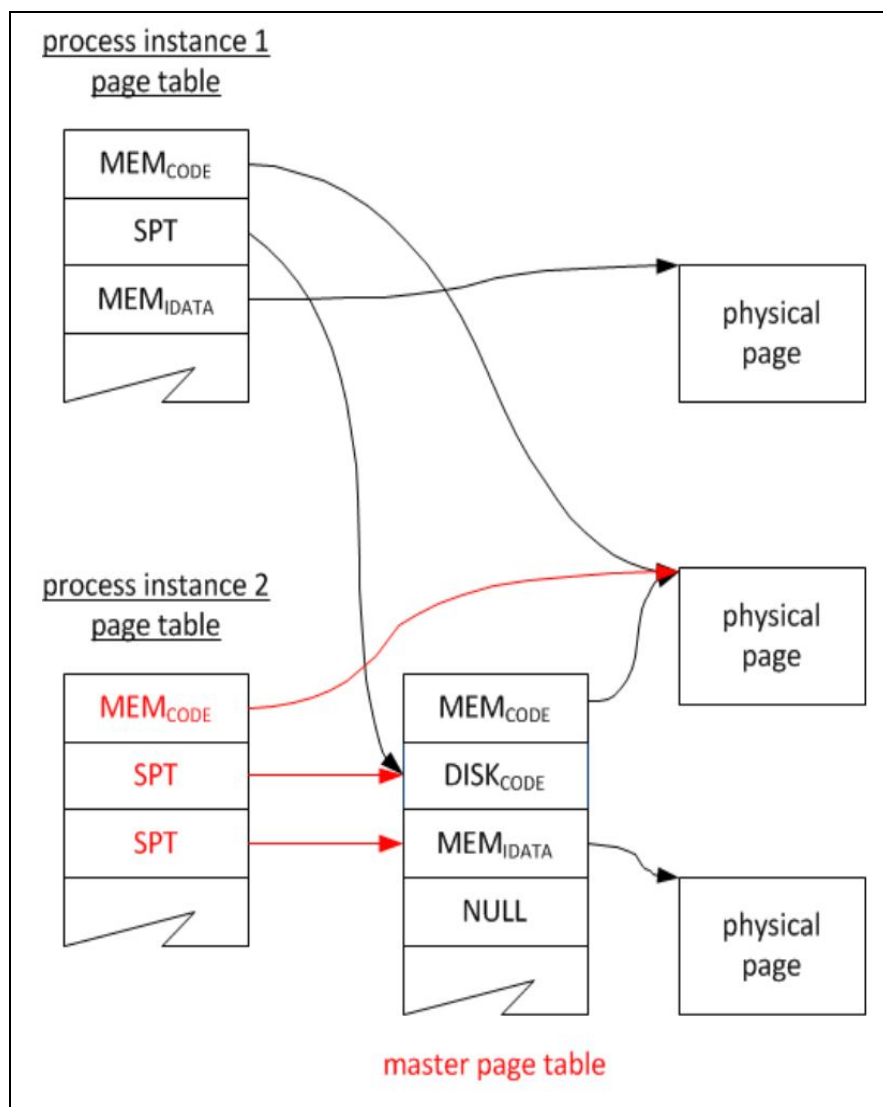


DISK_{CODE} - Allocate page of physical memory. Fill with data read from disk. Update PTEs in master and process page tables to point to the allocated page [**MEM_{CODE}**]

DISK_{IDATA} - Allocate page of physical memory (1). Fill with data read from disk. Attach to master page table [**MEM_{IDATA}**]. Now have read-only copy of initialised data.

Then, allocate page of physical memory (2). Copy data from master copy (1). Attach to process page table [**MEM_{IDATA}**]. Process now has its own copy of the initialised data which it is free to overwrite.





Once these page tables have been mapped and are referenced correctly in the master page table it is extremely easy for another instance of a process to be brought up.

The **MEM_{CODE}** entries are copied thus sharing the code. The remaining PTE's for the code and init data are set to SPT and point to corresponding entry in the master page table. The remaining PTEs are initialised as per non-shared case as every process needs its own stack and heap.

If a process is **killed** the OS will try to keep the master table and its attached pages in memory. This way, when another instance of the process is created it can quickly attach to the code pages already in memory and boot up a lot faster.