# Lab 8: Page Table Walker in PyRTL

| | |
|---|---|
| **Assigned**: | *Wednesday, February 28th, 2024* |
| **Due**: | *Wednesday, March 6th, 2024* |
| **Points**: | *100* |

• MAY ONLY BE TURNED IN ON **GRADESCOPE as PYTHON files** (see below for details).

• There is NO MAKEUP for missed assignments.

• We strictly enforce the LATE POLICY for all assignments (see syllabus).

## Goals for This Lab

During this lab you will learn how to build a Page Table walker in PyRTL. In doing so, you will learn the basics of Virtual Memory implementations in hardware.

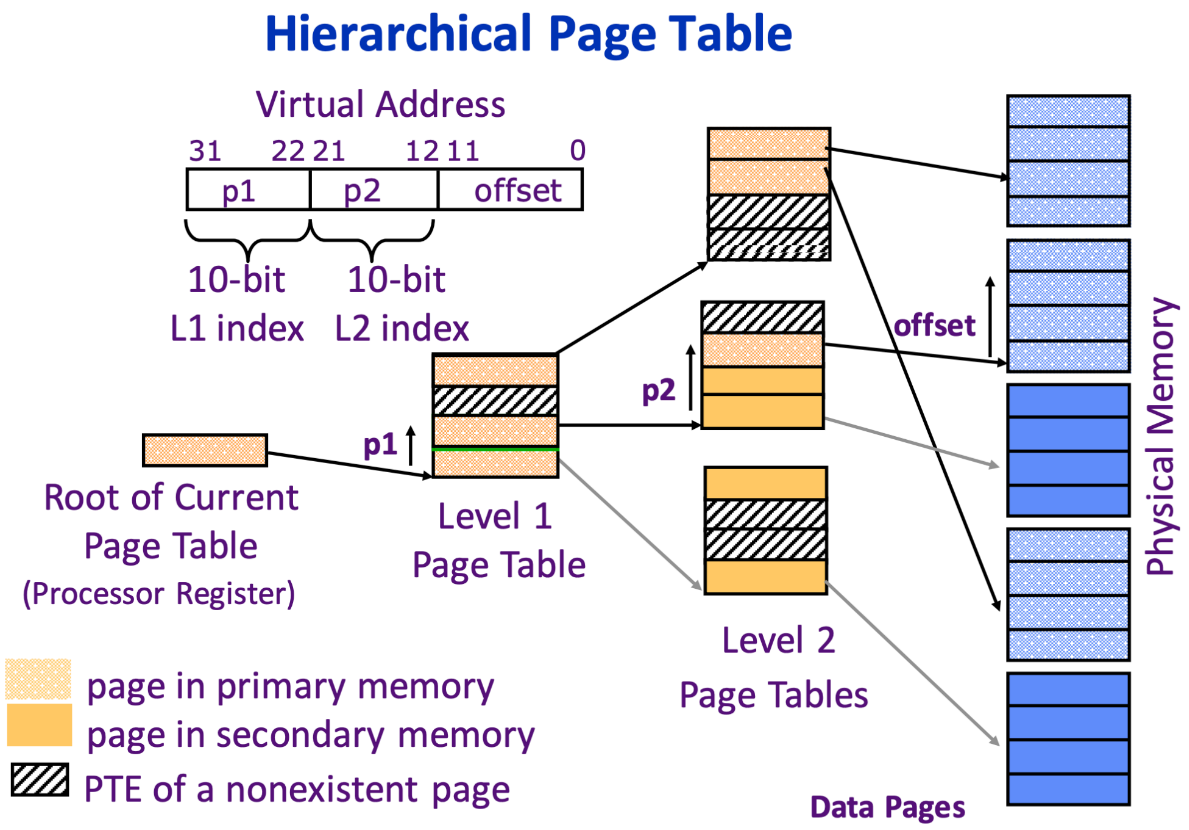


Figure 1: From the Lecture Slides on Page Table Walkers (via David Wentzlaff from Princeton). Notice how each entry in the page table will lead to a new page

## Task and Background

Your task is to implement a **2-level Page Table Walker**. Recall from class that virtual addresses used by programs are translated into physical addresses before accessing memory. All the translations are held in a data structure in memory called a page table. Using a page table with 1 level results in page tables that are too large. Instead, we use hierarchies of page tables (see the figure above). Each level's page table holds the offset into the next level's page table until you reach the final level where the physical page numbers are located. A page table walker is a piece of hardware which traverses this hierarchy. The traversal of this hierarchy takes **two** memory accesses.

## Provided Files

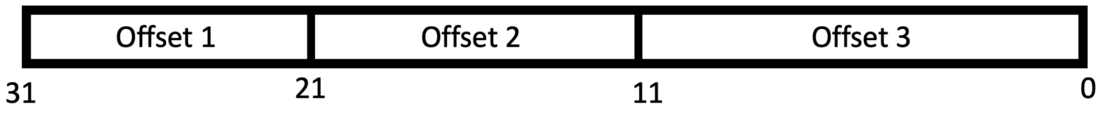
We have provided the skeleton file ucsbcs154lab8_ptw.py, where you will need to implement your page table walker. You will upload the solution to gradescope as ucsbcs154lab8_ptw.py

## Inputs and Outputs

### Inputs:

A large part of this lab involves decoding input signals. There are 4 input signals:

"virtual_addr": This is the virtual address. The virtual address should be split into three components shown below:



Each offset corresponds to the offset from the page location to get to the desired entry. The offsets are listed in order of usage, so offset 1 is the offset to use in the first step of the walk, offset 2 in the second step, and offset 3 to get the physical address.

"new_req": 1 if there is an incoming request for translation, 0 otherwise in which case you should do nothing.

"reset" : If 1, then reset the state back to 0 (stop translating)

"req_type" : 0 if the request is a read, 1 if the request is a write

## Outputs:

"physical_addr": the translated address signal ONLY WHEN THE PAGE WALK IS FINISHED (until then it should stay 0x0)

"dirty": 1 if the page table had the dirty bit enabled

"valid": 1 if the page table had the valid bit enabled

"ref": 1 if the page table had the ref bit enabled

"finished_walk": 1 if the walk has finished, i.e. the physical address is ready

"error_code": one of four options dictated by the table below:

| Error Code | Meaning | Stages it can be seen on |
|---|---|---|
| 000 | No errors | States 0b00 0b01 0b10 |
| 001 | Page Fault during walk | States 0b00 0b01 0b10 |
| 010 | Trying to write entry which is not writeable | State 0b10 |
| 100 | Trying to read entry which is not readable | State 0b10 |

## Other Important Signals/Pieces:

"state" : current state of the page table walker (more to be discussed below)

"page_fault" : You are not required to output this signal directly but it can help in simplifying your logic

"base_register": the entry point for the page table walker. (Note: even though it is called a register, we're using a Constant value to make it easier to use)

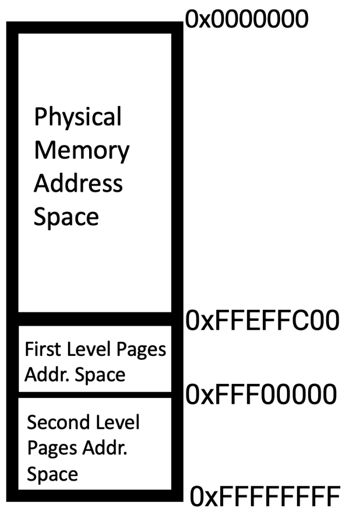
"main_memory" : Main memory is where all the pages are held. It has a specific structure that is described below. We give you the entry point, so the specifics are not important to the implementation of this lab but nonetheless are helpful to know.

## Structure of Memory for this Lab

Memory is once again word addressable in this lab. This means that for each index, there are a corresponding 32 bits.

The diagram to the right shows the structure of the main memory. As you can see, the physical memory space takes up a vast majority of the main memory. The page tables take up a small part of the memory at the high end of the address space.

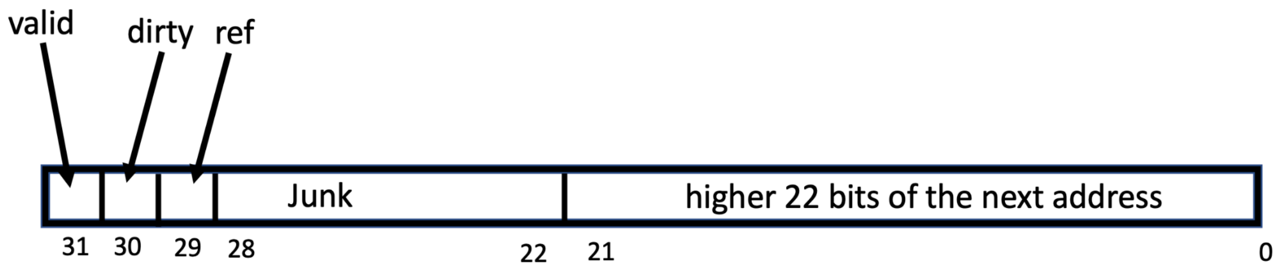
The base register is the higher 22 bits of where the first page is.


0x0000000
Physical Memory Address Space
0xFFEFFC00
First Level Pages Addr. Space
0xFFF00000
Second Level Pages Addr. Space
0xFFFFFFFF

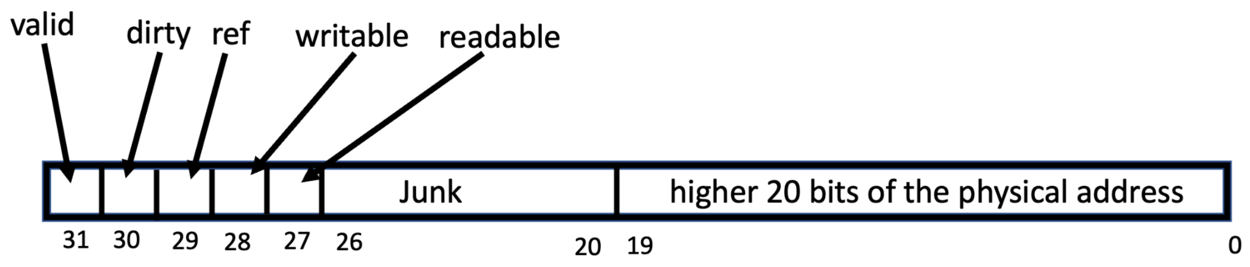
## Page Table Structure

Each page table has 1024 entries, so each table starts on a 10-bit aligned address (this is why the first level pages start at 0xFFEFFC00, so that the last 10 bits are all 0).

Each entry in the first level has the following structure:



Each entry in the second level has the following structure:



The bits in the "next address" section of each entry are the high order bits and concatenated with the corresponding offset in the *virtual_addr* input to form the full address for the next entry.

## Example of a Page Table Walk (This is the test given in the template)

The input to the page table walker is

0b11010000001110001000110110110011 (= 0xD0388DB3)

We can split this up into the appropriate offsets:

Offset 1: 1101000000 (= 0x340)

Offset 2: 1110001000 (= 0x388)

Offset 3: 110110110011 (= 0xDB3)

The base register holds the binary value 11 1111 1111 1011 1111 1111 (= 0x3FFBFF)

The first step is to concatenate the base register to offset 1, giving the following address:

11111111111011111111111101000000 (= 0xFFEFFF40)

Now we can go to main memory at that address, which contains (for this example) a page table entry with the following value:

11000100001111111111110001101011 (= 0xC43FFC6B)

Since the valid bit is set, there is no page fault and we can extract the higher 22 bits of the next address

1111111111110001101011 (= 0x3FFC6B)

We can concatenate this to offset 2 to get the following address:

11111111111100011010111110001000 (= 0xFFF1AF88)

Now we can go to main memory at that address, which contains (for this example) a page table entry with the following value:

10101100000001100001110100100110 (= 0xAC061D26)

Since there is no page fault, we can construct the final, physical address by taking the last 20 bits and concatenating them to offset 3, giving a final physical address of

01100001110100100110110110110011 (= 0x61D26DB3)

And giving an error code of 000

## Determining a Page Fault

It's important to return when a page fault occurs. A page fault occurs when two conditions are met:

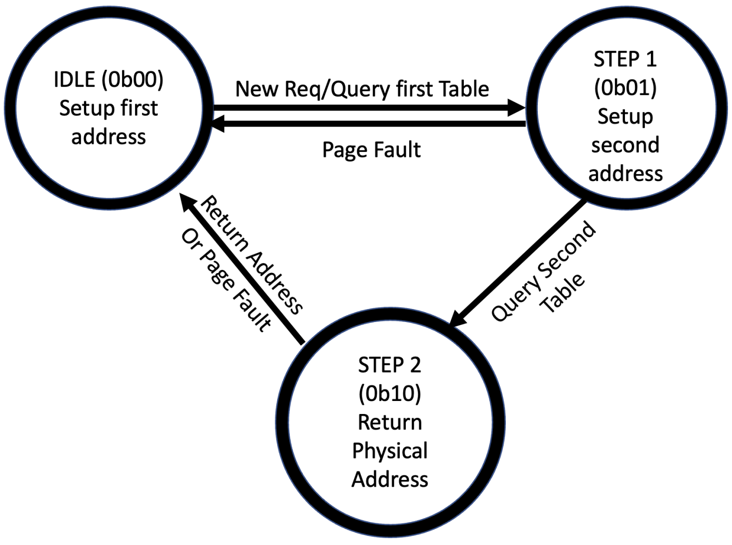
- You're not in the idle state
- The valid bit of the table you queried is false

## State Machine for controlling the walk

The intended way to implement the page table is such that it will take exactly 3 cycles to execute (if it takes any less or any more, then the autograder might fail your design). Each cycle can be seen as a state in a state machine. We start in an idle state and when we get a new request we transition into the page table walk.

A state machine will tell your hardware when it's starting the walk, what stage it is in the walk, and when to stop the walk and return the data.

A diagram of the state machine can be seen below:


IDLE (0b00) Setup first address
New Req/Query first Table
Page Fault
STEP 1 (0b01) Setup second address
Query Second Table
STEP 2 (0b10) Return Physical Address
Return Address Or Page Fault

Some points worth mentioning, even if you haven't finished the walk, a page fault will send you back to the IDLE State. Additionally, in the IDLE state, you are still setting up the first address to query on, that way, in the second state (0b01) you can already be setting up the next address.

## Test your Design!

We have provided you with an environment in which you may test your page table walker. We have provided you with some barebones test cases. Within the main function of the skeleton files, you will see code that will perform the example given in the document above. ***Note: these tests are NOT comprehensive, and you should write additional tests independently to check for edge cases.***

Make sure to test for the following edge cases!:

- Page Faults (i.e. error code 001)
- Reset Signals (i.e. if the reset signal is active, you should not be outputting anything but 0s)
- Readability (i.e. error code 100)
- Writability (i.e. error code 010)

## Files to Submit

For this lab, you simply need to submit ucsbcs154lab8_ptw.py. Please double-check that you are not including Python imports at the top of your file unrelated to this lab (e.g., tkinter). These imports may prevent the autograder from running your submission.

Please keep your eyes open for any Piazza announcements in case we make updates to the submission requirements.

## Autograder

The autograder will reveal the name of the test, but we will NOT be releasing the actual test cases themselves. Part of this assignment is to learn how to test your code! **For this lab, you may not share the test code with other students in the course.** You may discuss testing strategy in terms of behaviors but not the specifics of the trace of memory access addresses. We want you to think through and develop an understanding of virtual to physical memory mapping.