MicroWatt Microarchitecture

H. Peter Hofstee
(uArch overview largely based on a slide deck by Paul Mackerras)

Contents

- MicroWatt Microarchitecture
- Adding new Instructions to MicroWatt

Goals for today:

- Intro to pipelining
- Learn more about Power instructions/formats & instruction recode/decode
- Intro to branch prediction
- Intro to cache design
- Show some vhdl

Next Tue (IST)

- Front end topics: Branch Prediction, Register Renaming, Dependency Checking

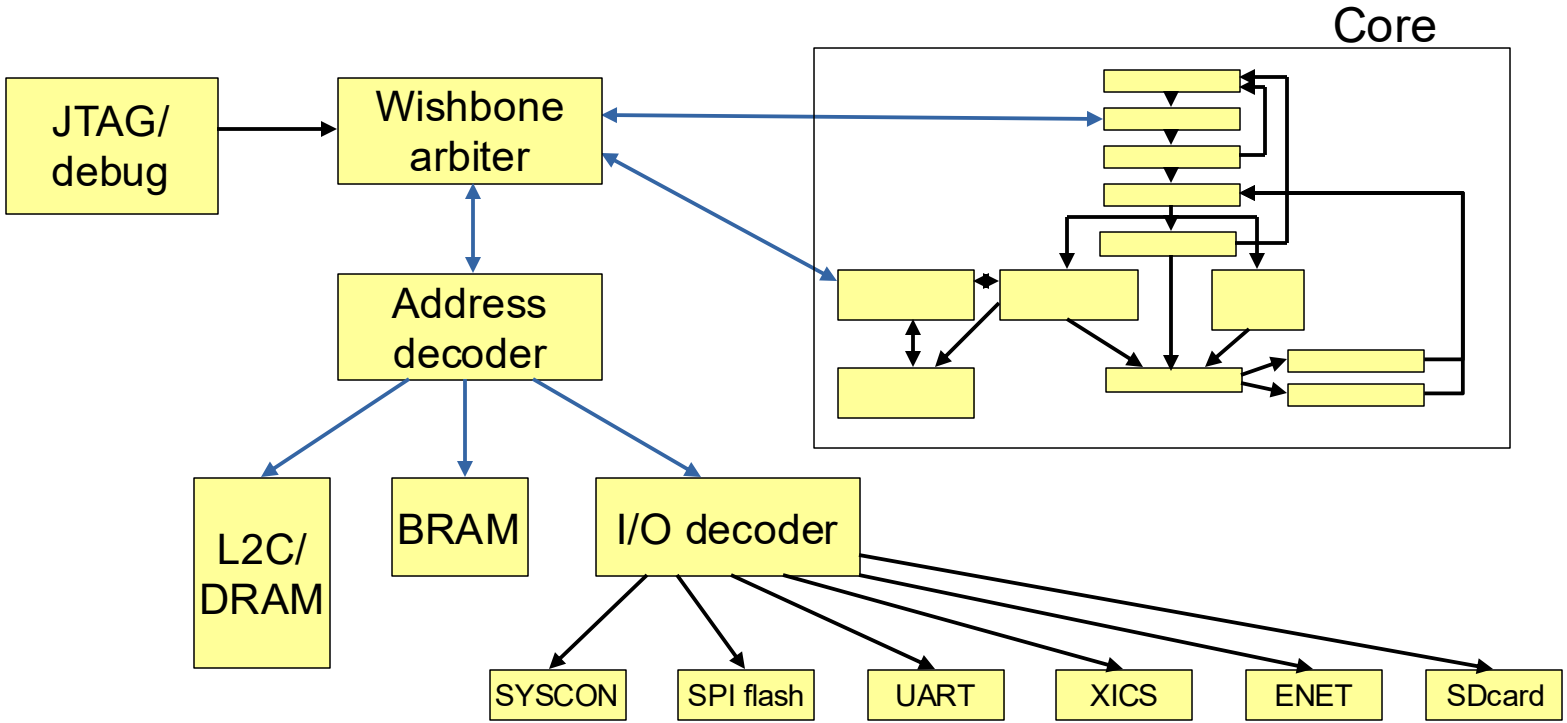
Next Th (IST)

- Back end topics: Load-Store Unit, Instruction Writeback and Completion

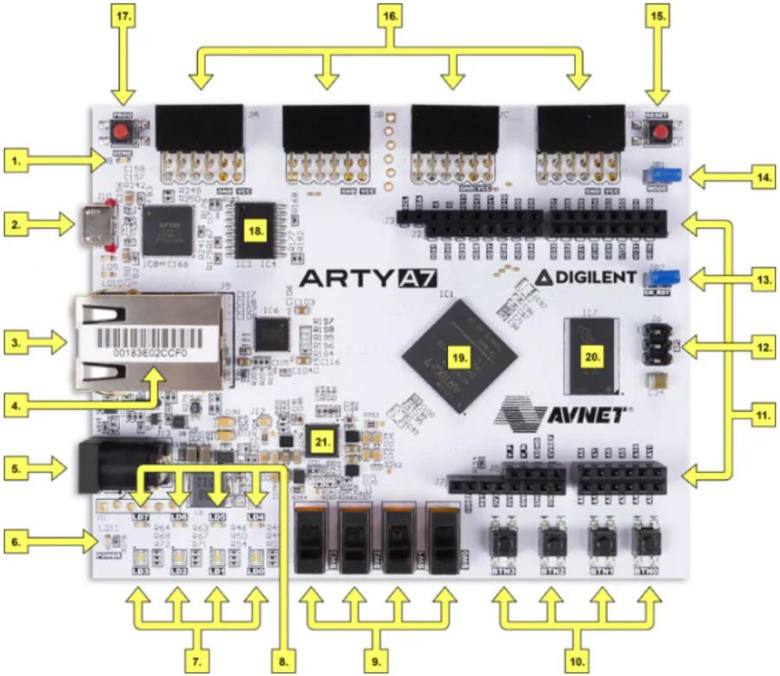
Introduction

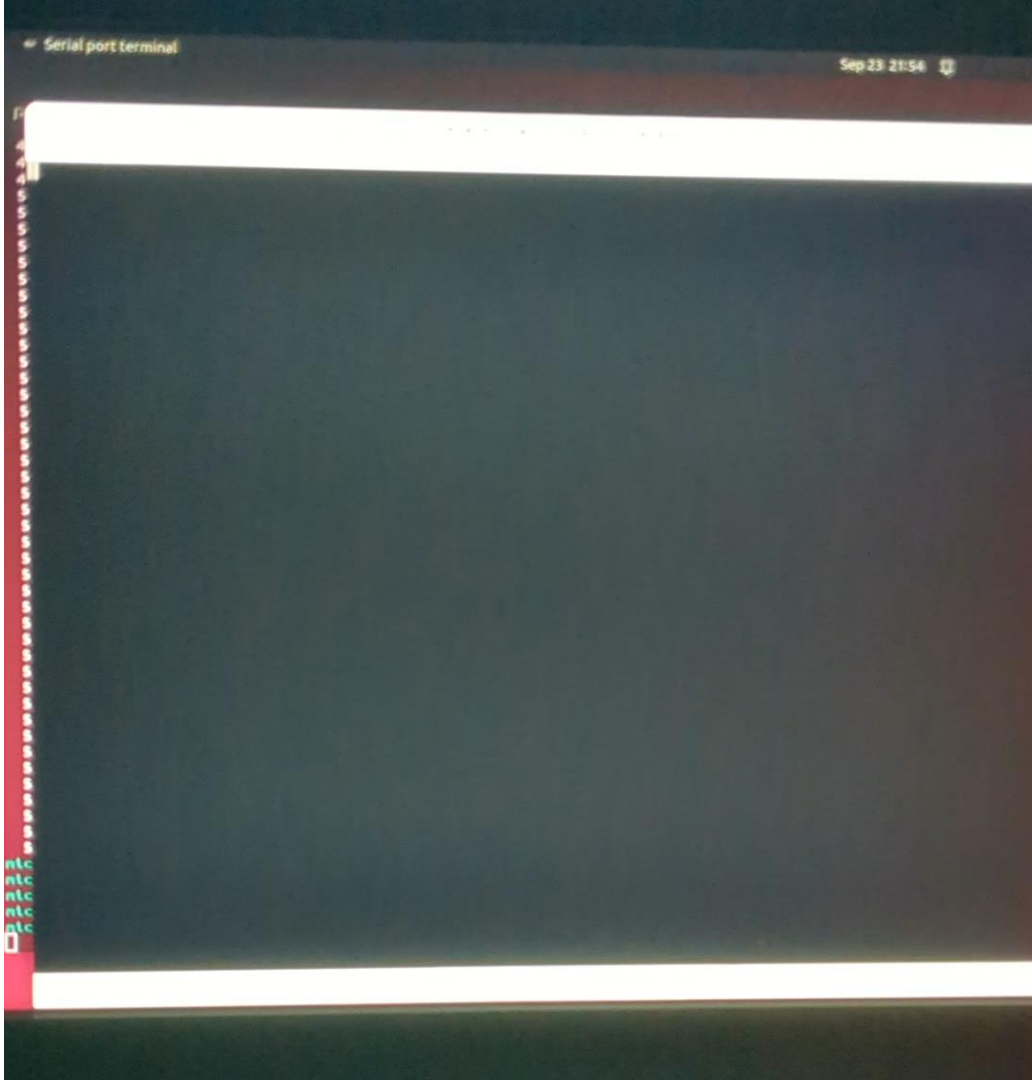
- Microwatt started as demo/proof-of-concept for announcement of Power ISA being made open (August 2019).
  - Core structure is relatively simple
- Code hosted on github, updated through pull requests
  - https://github.com/antonblanchard/microwatt
  - Use automated testing to catch bugs early
- Aims to be a compliant PowerISA v3.1C implementation
  - Includes support for prefixed instructions (2-word instructions)
- Can run Linux
- Targets simulation and synthesis for FPGAs
  - E.g. Artix-7, see https://github.com/hofstee-hp/MicroWatt_on_Arty_A7-100T
- Wishbone interface to memory (and memory-mapped peripherals)
- Peripherals from Litex project

System overview

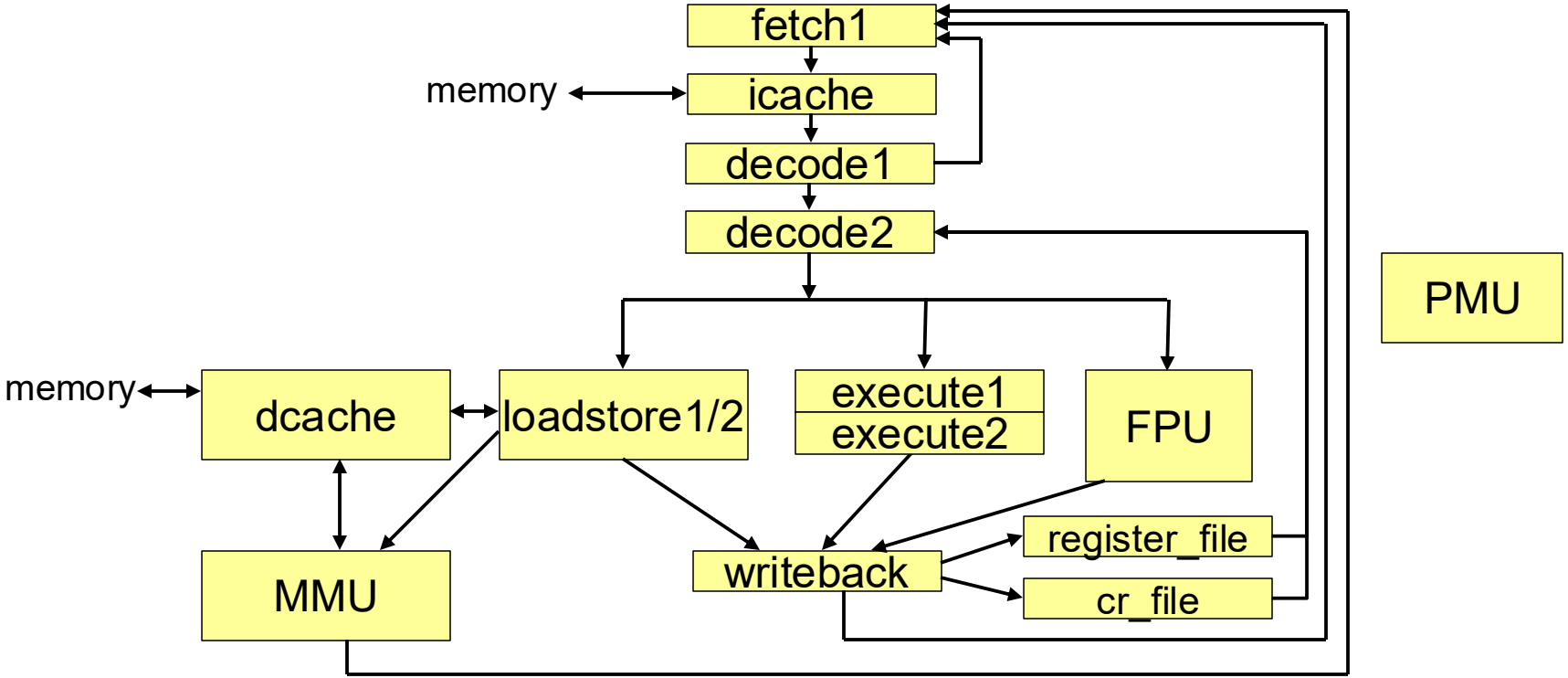


Arty A7-100T


17.
16.
15.
1.
2.
3.
4.
5.
6.
14.
13.
12.
11.
18.
19.
20.
21.
7.
8.
9.
10.
ARTY A7
DIGILENT
AVNET


Serial port terminal
Sep 23 21:54

Pipeline Overview



Fetch1 and icache

- Fetch1 generates a sequence of effective and corresponding real addresses
  - Increment by 4 each cycle unless redirected by decode1 or writeback
    - Branches, interrupts, rfid, isync cause redirects
  - Contains instruction TLB and tiny 2-entry cache of TLB
  - Contains (optional) Branch Target Cache (BTC)
    - Stores instruction address, target address and taken indication for executed direct branches
- Icache reads and caches instructions from memory
  - Use of real address for index and tag avoids aliasing problems
  - Instructions are predecoded into 36-bit form on icache refill
    - 6-bit primary opcode gets replaced by 10-bit instruction index determined from primary and extended opcodes
  - No support currently from fetching from cache-disabled pages of memory
  - Icache snoops writes to memory and invalidates corresponding lines

Instruction Opcode Recoding (predecode.vhdl)

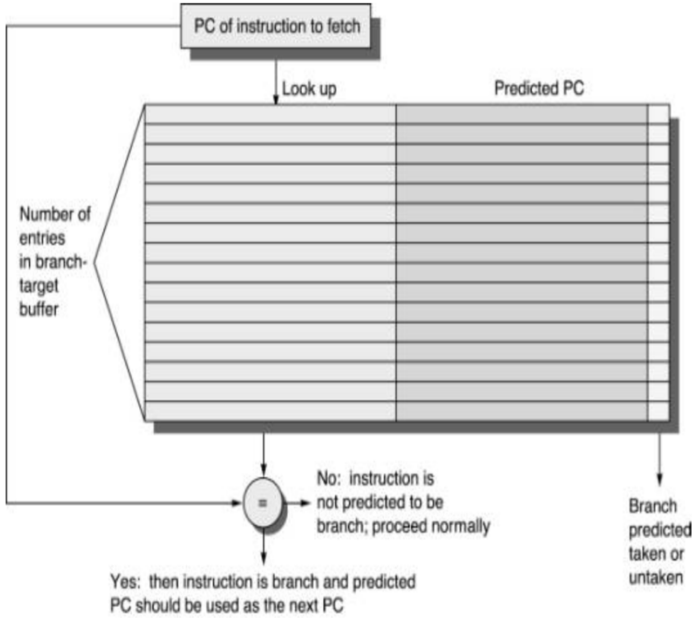
- OpenPOWER has MANY instruction formats
  - pp. 11-16 of v3.1c ISA document
- All have a 6 bit primary opcode (PO)
  - Extended opcodes (XO) range from 0 to 11 additional bits
  - Sometimes other modifier bits also effectively change the operation
- Many POWER processors recode instruction bits before storing in the iCache for easier decode in the rest of the pipeline
  - Not only opcodes (as in microwatt) some also regularize register operand locations etc.

```
architecture behaviour of predecoder is

    type predecoder_rom_t is array(0 to 2047) of insn_code;

    constant major_predecode_rom : predecoder_rom_t := (
        2#001100_00000# to 2#001100_11111# =>  INSN_addic,
        2#001101_00000# to 2#001101_11111# =>  INSN_addic_dot,
        2#001110_00000# to 2#001110_11111# =>  INSN_addi,
        2#001111_00000# to 2#001111_11111# =>  INSN_addis,
        2#010011_00100# to 2#010011_00101# =>  INSN_addpcis,
        2#011100_00000# to 2#011100_11111# =>  INSN_andi_dot,
        2#011101_00000# to 2#011101_11111# =>  INSN_andis_dot,
        2#000000_00000#                    =>  INSN_attn,
        2#010010_00000# to 2#010010_00001# =>  INSN_brel,
        2#010010_00010# to 2#010010_00011# =>  INSN_babs,
        2#010010_00100# to 2#010010_00101# =>  INSN_brel,
        2#010010_00110# to 2#010010_00111# =>  INSN_babs,
        2#010010_01000# to 2#010010_01001# =>  INSN_brel,
        2#010010_01010# to 2#010010_01011# =>  INSN_babs,
        2#010010_01100# to 2#010010_01101# =>  INSN_brel,
        2#010010_01110# to 2#010010_01111# =>  INSN_babs,
        2#010010_10000# to 2#010010_10001# =>  INSN_brel,
        2#010010_10010# to 2#010010_10011# =>  INSN_babs,
        2#010010_10100# to 2#010010_10101# =>  INSN_brel,
        2#010010_10110# to 2#010010_10111# =>  INSN_babs,
        2#010010_11000# to 2#010010_11001# =>  INSN_brel,
        2#010010_11010# to 2#010010_11011# =>  INSN_babs,
        2#010010_11100# to 2#010010_11101# =>  INSN_brel,
        2#010010_11110# to 2#010010_11111# =>  INSN_babs,
```

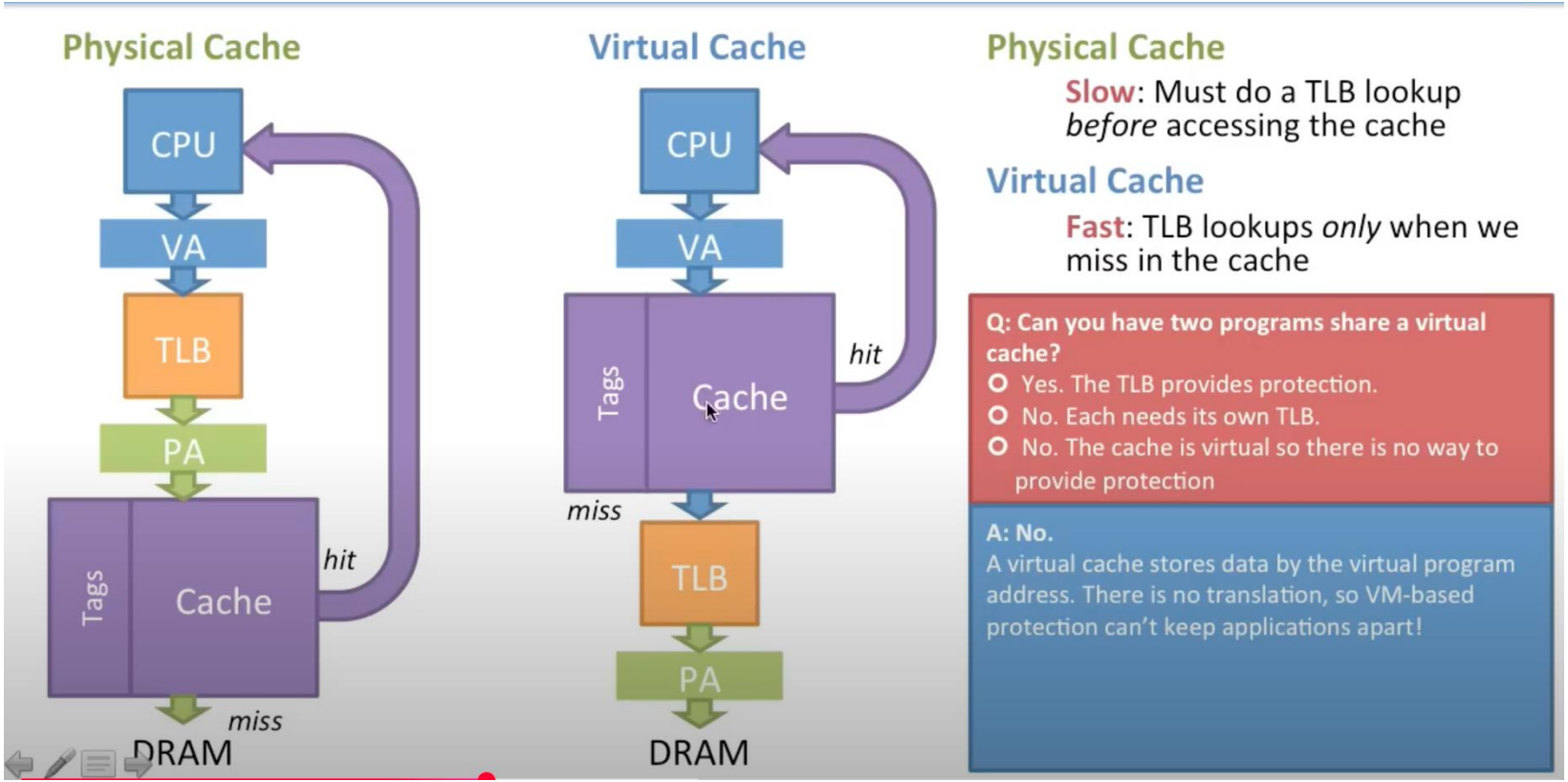
Branch Target Address Cache (Fetch1)




```
-- If there is a valid entry in the BTC which corresponds to the next instruction,
-- use that to predict the address of the instruction after that.
-- (w_in.redirect = '0' and d_in.redirect = '0' and r_int.tlbstall = '0')
-- implies v.nia = r_int.next_nia.
-- r_int.rd_is_niap4 implies r_int.next_nia is the address used to read the BTC.
if v.req = '1' and w_in.redirect = '0' and d_in.redirect = '0' and r_int.tlbstall = '0' and
        btc_rd_valid = '1' and r_int.rd_is_niap4 = '1' and
        btc_rd_data(BTC_WIDTH - 2) = r.virt_mode and
        btc_rd_data(BTC_WIDTH - 3 downto BTC_TARGET_BITS)
            = r_int.next_nia(BTC_TAG_BITS + BTC_ADDR_BITS + 1 downto BTC_ADDR_BITS + 2) then
    v.predicted := btc_rd_data(BTC_WIDTH - 1);
    v.pred_ntaken := not btc_rd_data(BTC_WIDTH - 1);
    if btc_rd_data(BTC_WIDTH - 1) = '1' then
        v_int.next_nia := btc_rd_data(BTC_TARGET_BITS - 1 downto 0) & "00";
        v_int.rd_is_niap4 := '0';
    end if;
end if;
```

https://superchargedcomputing.com/2018/02/23/modern-superscalar-processors/

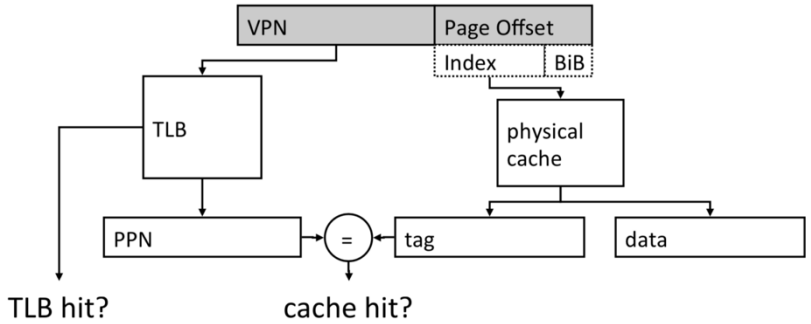
Virtually vs. Physically Addressed Cache



Source: https://www.youtube.com/watch?v=3sX5obQCHNA

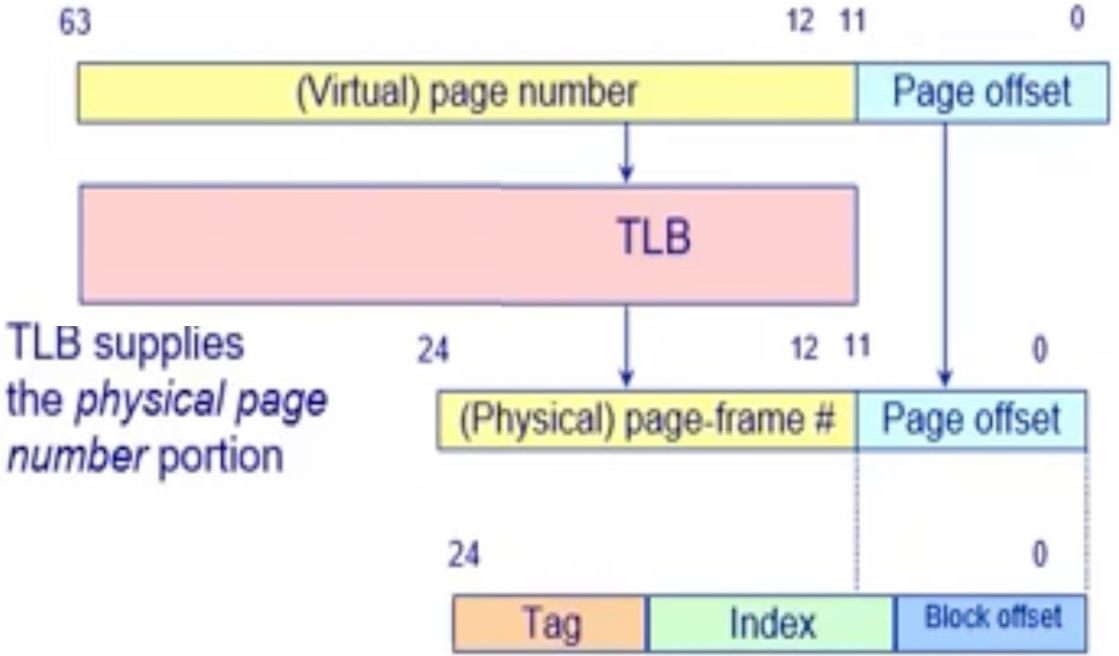
Basic Cache Types

- Physically Indexed, Physically Tagged (PIPT)
  - Straightforward, but slow, because address translation has to be done first
  - Microwatt iCache design keeps last two TLB entries
- Virtually Indexed, Virtually Tagged
  - Fast (no translation required), but …
  - ….need to clear on context switch (because of security and aliasing issues)
- Virtually Indexed, Physically Tagged (VIPT)
  - To do translation in parallel w. cach access each set is limited to the page size

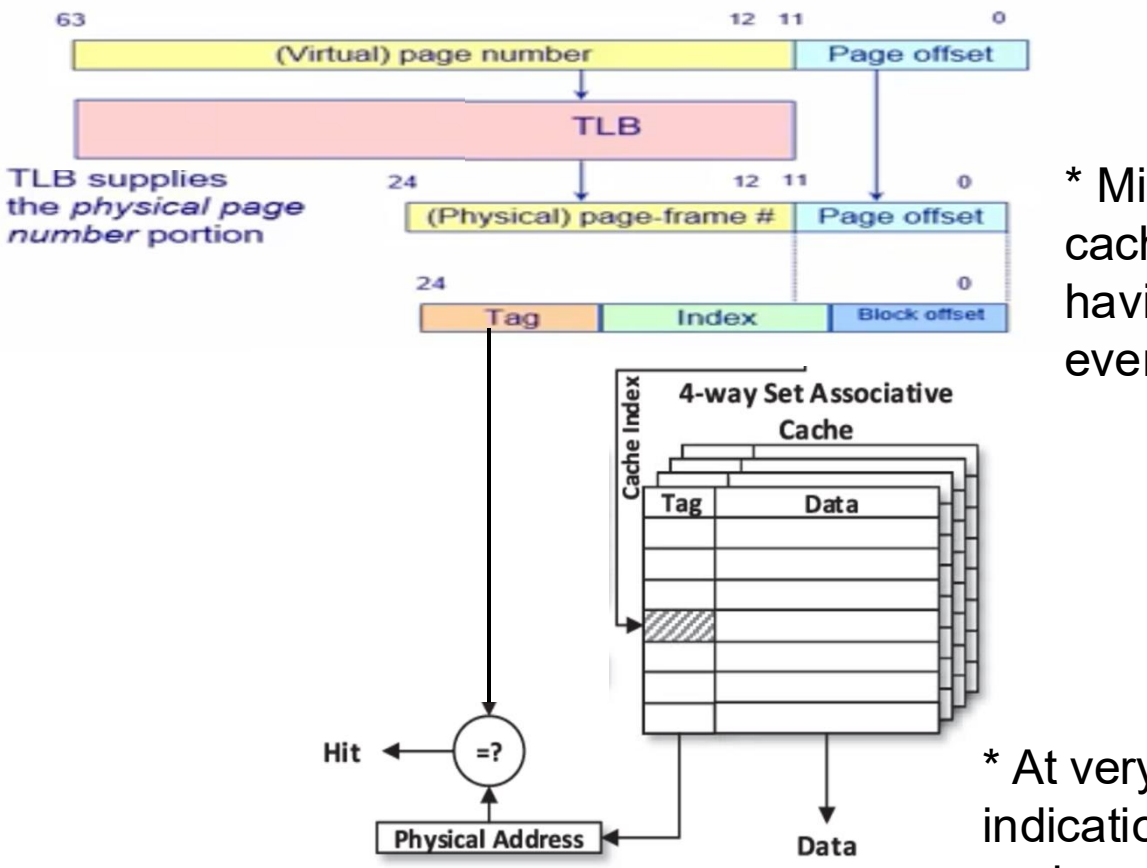


http://thebeardsage.com/virtual-memory-cache-tlb-interaction/

Virtual vs. Real/Physical Address



Cache Access



* Microwatt uses a 2-entry cache of the TLB to avoid having to access the TLB for every iFetch.

* At very least also need valid indication per entry (also replacement-related bits).

Decode1 and decode2

- Decode1 looks up a decode ROM using instruction index
  - Decode ROM gives control signals used to control operand decoding, ALU function, result selection, etc.
  - Does static branch prediction for direct branches
  - Computes register file addresses (GPRs and FPRs) for up to 3 register operands
  - Decodes SPR (special purpose register) numbers
- Decode2 does instruction scheduling and dispatch
  - Calculates instruction dependencies and stalls until operands are available
  - Forwards operands from previous instruction results (where available)
  - Does immediate operand selection and formatting
    - Operand can come from instruction field, or be the address of the instruction
  - Computes some control signals for execute stage

Decode1 decoder ROM

```
type decoder_rom_t is array(insn_code) of decode_rom_t;

constant decode_rom : decoder_rom_t := (
    --                  unit   fac   internal      in1         in2  const       in3   out   CR   CR   inv  inv  cry   cry  ldst  BR   sgn  upd  rsrv 32b  sgn  rc    lk   priv sgl  rpt
    --                                   op                                                  in   out  A    out  in    out  len        ext                                    pipe
    INSN_illegal    =>  (ALU,  NONE, OP_ILLEGAL,   NONE,       IMM, NONE,       NONE, NONE, '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
    INSN_fetch_fail =>  (LDST, NONE, OP_FETCH_FAILED, CIA,     IMM, NONE,       NONE, NONE, '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),

    INSN_add        =>  (ALU,  NONE, OP_ADD,       RA,         RB,  NONE,       NONE, RT,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', RCOE, '0', '0', '0', NONE),
    INSN_addc       =>  (ALU,  NONE, OP_ADD,       RA,         RB,  NONE,       NONE, RT,   '0', '0', '0', '0', ZERO, '1', NONE, '0', '0', '0', '0', '0', '0', RCOE, '0', '0', '0', NONE),
    INSN_adde       =>  (ALU,  NONE, OP_ADD,       RA,         RB,  NONE,       NONE, RT,   '0', '0', '0', '0', CA,   '1', NONE, '0', '0', '0', '0', '0', '0', RCOE, '0', '0', '0', NONE),
    INSN_addex      =>  (ALU,  NONE, OP_ADD,       RA,         RB,  NONE,       NONE, RT,   '0', '0', '0', '0', OV,   '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0', '0', NONE),
    INSN_addg6s     =>  (ALU,  NONE, OP_ADDG6S,    RA,         RB,  NONE,       NONE, RT,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
    INSN_addi       =>  (ALU,  NONE, OP_ADD,       RA_OR_ZERO, IMM, CONST_SI,   NONE, RT,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
    INSN_addic      =>  (ALU,  NONE, OP_ADD,       RA,         IMM, CONST_SI,   NONE, RT,   '0', '0', '0', '0', ZERO, '1', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
    INSN_addic_dot  =>  (ALU,  NONE, OP_ADD,       RA,         IMM, CONST_SI,   NONE, RT,   '0', '0', '0', '0', ZERO, '1', NONE, '0', '0', '0', '0', '0', '0', ONE,  '0', '0', '0', NONE),
    INSN_addis      =>  (ALU,  NONE, OP_ADD,       RA_OR_ZERO, IMM, CONST_SI_HI, NONE, RT,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
    INSN_addme      =>  (ALU,  NONE, OP_ADD,       RA,         IMM, CONST_M1,   NONE, RT,   '0', '0', '0', '0', CA,   '1', NONE, '0', '0', '0', '0', '0', '0', RCOE, '0', '0', '0', NONE),
    INSN_addpcis    =>  (ALU,  NONE, OP_ADD,       CIA,        IMM, CONST_DXHI4, NONE, RT,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
    INSN_addze      =>  (ALU,  NONE, OP_ADD,       RA,         IMM, NONE,       NONE, RT,   '0', '0', '0', '0', CA,   '1', NONE, '0', '0', '0', '0', '0', '0', RCOE, '0', '0', '0', NONE),
    INSN_and        =>  (ALU,  NONE, OP_LOGIC,     NONE,       RB,  NONE,       RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0', '0', NONE),
    INSN_andc       =>  (ALU,  NONE, OP_LOGIC,     NONE,       RB,  NONE,       RS,   RA,   '0', '0', '1', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0', '0', NONE),
    INSN_andi_dot   =>  (ALU,  NONE, OP_LOGIC,     NONE,       IMM, CONST_UI,   RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', ONE,  '0', '0', '0', NONE),
    INSN_andis_dot  =>  (ALU,  NONE, OP_LOGIC,     NONE,       IMM, CONST_UI_HI, RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', ONE,  '0', '0', '0', NONE),
    INSN_attn       =>  (ALU,  NONE, OP_ATTN,      NONE,       IMM, NONE,       NONE, NONE, '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '1', '1', NONE),
    INSN_brel       =>  (ALU,  NONE, OP_B,         CIA,        IMM, CONST_LI,   NONE, NONE, '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '1', '0', '0', NONE),
    INSN_babs       =>  (ALU,  NONE, OP_B,         NONE,       IMM, CONST_LI,   NONE, NONE, '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '1', '0', '0', NONE),
    INSN_bcrel      =>  (ALU,  NONE, OP_BC,        CIA,        IMM, CONST_BD,   NONE, NONE, '1', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '1', '0', '0', NONE),
    INSN_bcabs      =>  (ALU,  NONE, OP_BC,        NONE,       IMM, CONST_BD,   NONE, NONE, '1', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '1', '0', '0', NONE),
    INSN_bcctr      =>  (ALU,  NONE, OP_BCREG,     NONE,       IMM, NONE,       NONE, NONE, '1', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '1', '0', '0', NONE),
    INSN_bclr       =>  (ALU,  NONE, OP_BCREG,     NONE,       IMM, NONE,       NONE, NONE, '1', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '1', '0', '0', NONE),
    INSN_bctar      =>  (ALU,  NONE, OP_BCREG,     NONE,       IMM, NONE,       NONE, NONE, '1', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '1', '0', '0', NONE),
    INSN_bperm      =>  (ALU,  NONE, OP_BPERM,     NONE,       RB,  NONE,       RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
    INSN_brh        =>  (ALU,  NONE, OP_BREV,      NONE,       IMM, NONE,       RS,   RA,   '0', '0', '0', '0', ZERO, '0', is2B, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
    INSN_brw        =>  (ALU,  NONE, OP_BREV,      NONE,       IMM, NONE,       RS,   RA,   '0', '0', '0', '0', ZERO, '0', is4B, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
    INSN_brd        =>  (ALU,  NONE, OP_BREV,      NONE,       IMM, NONE,       RS,   RA,   '0', '0', '0', '0', ZERO, '0', is8B, '0', '0', '0', '0', '0', '0', NONE, '0', '0', '0', NONE),
```

Static Branch Predict (Decode1)

```
-- Branch predictor
-- Note bclr, bcctr and bctar not predicted as we have no
-- count cache or link stack.
br_offset := f_in.insn(25 downto 2);
case icode is
    when INSN_brel | INSN_babs =>
        -- Unconditional branches are always taken
        v.br_pred := '1';
    when INSN_bcrel =>
        -- Predict backward relative branches as taken, others as untaken
        v.br_pred := f_in.insn(15);
        br_offset(23 downto 14) := (others => '1');
    when others =>
end case;
```

Branch Conditional B-form

bc	BO,BI,target_addr	(AA=0 LK=0)
bca	BO,BI,target_addr	(AA=1 LK=0)
bcl	BO,BI,target_addr	(AA=0 LK=1)
bcla	BO,BI,target_addr	(AA=1 LK=1)

16	BO	BI	BD	AA	LK
0	6	11	16	30	31

```
if (64-bit mode)
   then M ← 0
   else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then
   if AA then NIA ←iea EXTS(BD || 0b00)
   else       NIA ←iea CIA + EXTS(BD || 0b00)
if LK then LR ←iea CIA + 4
```

Immediate Field Formatting (Decode2)

```
function decode_b_const (t : const_sel_t; insn_in : std_ulogic_vector(31 downto 0);
                         prefix : std_ulogic_vector(25 downto 0))
    return std_ulogic_vector is
    variable ret : std_ulogic_vector(63 downto 0);
begin
    case t is
        when CONST_UI =>
            ret := std_ulogic_vector(resize(unsigned(insn_ui(insn_in)), 64));
        when CONST_SI =>
            ret := std_ulogic_vector(resize(signed(insn_si(insn_in)), 64));
        when CONST_PSI =>
            ret := std_ulogic_vector(resize(signed(insn_prefixed_si(prefix, insn_in)), 64));
        when CONST_SI_HI =>
            ret := std_ulogic_vector(resize(signed(insn_si(insn_in)) & x"0000", 64));
        when CONST_UI_HI =>
            ret := std_ulogic_vector(resize(unsigned(insn_si(insn_in)) & x"0000", 64));
        when CONST_LI =>
            ret := std_ulogic_vector(resize(signed(insn_li(insn_in)) & "00", 64));
        when CONST_BD =>
            ret := std_ulogic_vector(resize(signed(insn_bd(insn_in)) & "00", 64));
        when CONST_DS =>
            ret := std_ulogic_vector(resize(signed(insn_ds(insn_in)) & "00", 64));
        when CONST_DQ =>
            ret := std_ulogic_vector(resize(signed(insn_dq(insn_in)) & "0000", 64));
        when CONST_DXHI4 =>
            ret := std_ulogic_vector(resize(signed(insn_dx(insn_in)) & x"0004", 64));
        when CONST_M1 =>
            ret := x"FFFFFFFFFFFFFFFF";
        when CONST_SH =>
            ret := x"00000000000000" & "00" & insn_in(1) & insn_in(15 downto 11);
        when CONST_SH32 =>
            ret := x"00000000000000" & "000" & insn_in(15 downto 11);
        when CONST_DSX =>
            ret := 55x"7FFFFFFFFFFFF" & insn_in(0) & insn_in(25 downto 21) & "000";
        when others =>
            ret := (others => '0');
    end case;

    return ret;
end;
```

- Many different ways in which a constant encoded in the instruction can turn into an operand.
- Decoder ROM in decode1 determines which variant applies.
- Decode2 calculates immediate values based on the instruction and the type of constant.

Execute1 and execute2

- Executes all instructions except load/store and FPU instructions
  - 64-bit integer multiply and integer division are now done by FPU (if present)
- Submodules handle shift/rotate instructions, logical instructions, count-bits instructions, 32-bit multiplications
- “Main adder” performs add/subtract instructions
- Exception detection and interrupt generation
- Interlocks to ensure instructions complete in order
- Execute1 does most integer computations and forwards results to decode2
  - Count-bits, multiply, and some move from SPR instructions take 2 cycles to generate their result, so their results are not forwarded from execute1
- Execute2 does updating of SPRs, condition-code result generation for dot-form instructions, and forwarding of results.

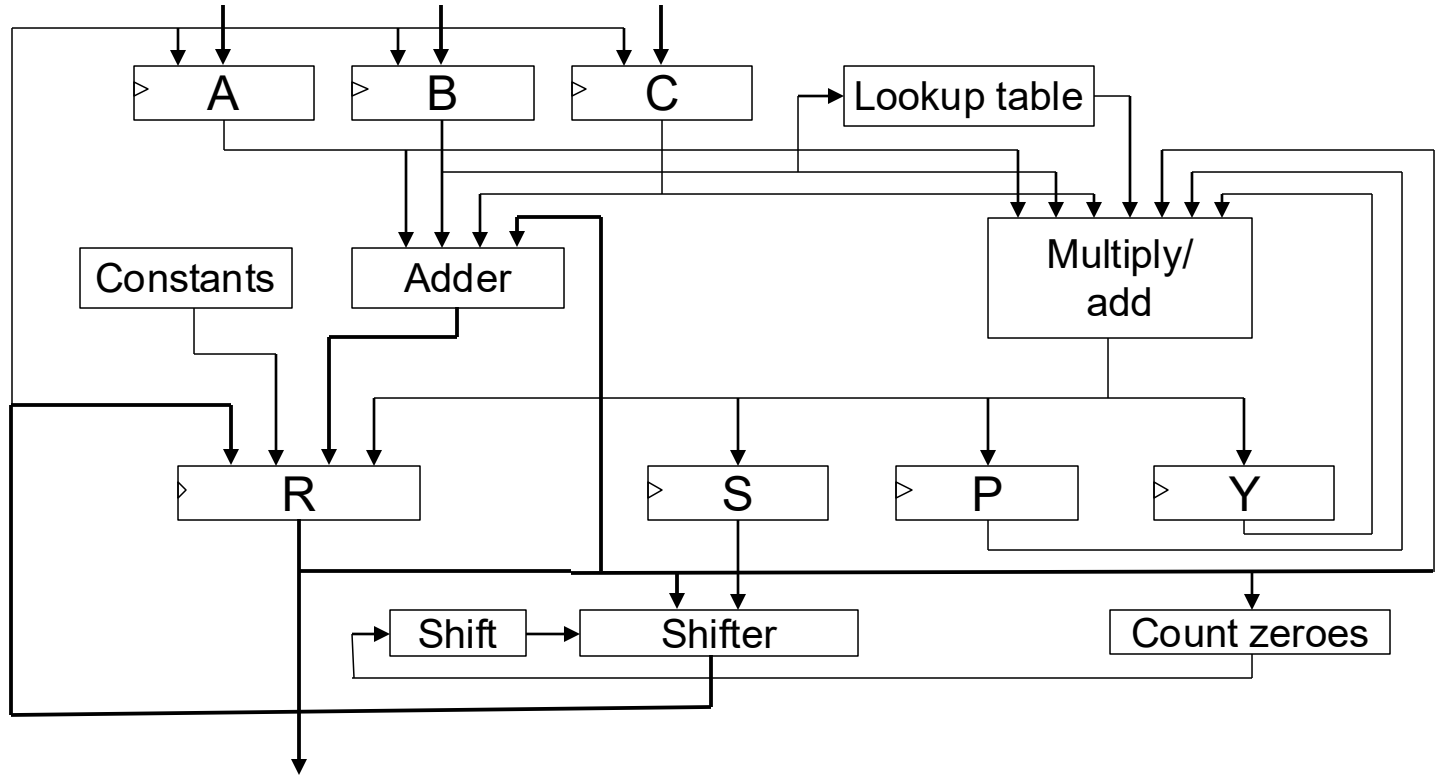
Loadstore1/2 and dcache

- Loadstore1 receives instruction from execute1
  - Evaluates whether the access is misaligned and crosses a doubleword boundary
  - Converts FP data to single-precision form for stfs[x] instruction
- Effective address sent to dcache in first cycle
- Dcache reads the cache RAM, cache tag, and TLB in the second cycle
  - Dcache is store-through; all stores are written to memory immediately
  - Cache tags and TLB looked up in parallel; both can have multiple ways
    - Matrix of tag comparators to decide on TLB and cache hits
  - State machine to handle stores, load misses and non-cacheable loads
  - Store data forwarded to later loads to avoid stalling load-hit-store cases
- For loads, formatting of result data occurs in third (writeback) cycle
  - Except conversion of single-precision data to DP form (lfs[x]) takes an extra cycle
- Loadstore 2 (and 3) has state machine to handle complex cases

Floating-point unit

- Non-pipelined
  - Has its own data path with 64-bit adder, shifter, multiplier (64x64 → 128) and count-leading-zeroes logic
- Control is a state machine
  - Implemented with “random” logic (not microcode)
  - One operation at a time
  - Add/subtract generally take 5–12 cycles, multiply takes 8–15 cycles
- Handles all architected exception conditions
  - Denormalized operands and results
  - Underflow and overflow conditions, including trap-enabled and trap-disabled results

FPU – Data paths (simplified)



PMU

- Used for performance monitoring
- Receives information from/via the execute1 stage
- Maintains various counters dependent on MMCR settings
- Can raise an interrupt for certain events

```
type PMUToExecute1Type is record
    spr_val : std_ulogic_vector(63 downto 0);
    intr    : std_ulogic;
end record;
```

```
type Execute1ToPMUType is record
    mfspr   : std_ulogic;
    mtspr   : std_ulogic;
    spr_num : std_ulogic_vector(4 downto 0);
    spr_val : std_ulogic_vector(63 downto 0);
    tbbits  : std_ulogic_vector(3 downto 0);
    pmm_msr : std_ulogic;
    pr_msr  : std_ulogic;
    run     : std_ulogic;
    nia     : std_ulogic_vector(63 downto 0);
    addr    : std_ulogic_vector(63 downto 0);
    addr_v  : std_ulogic;
    trace   : std_ulogic;
    occur   : PMUEventType;
end record;
```

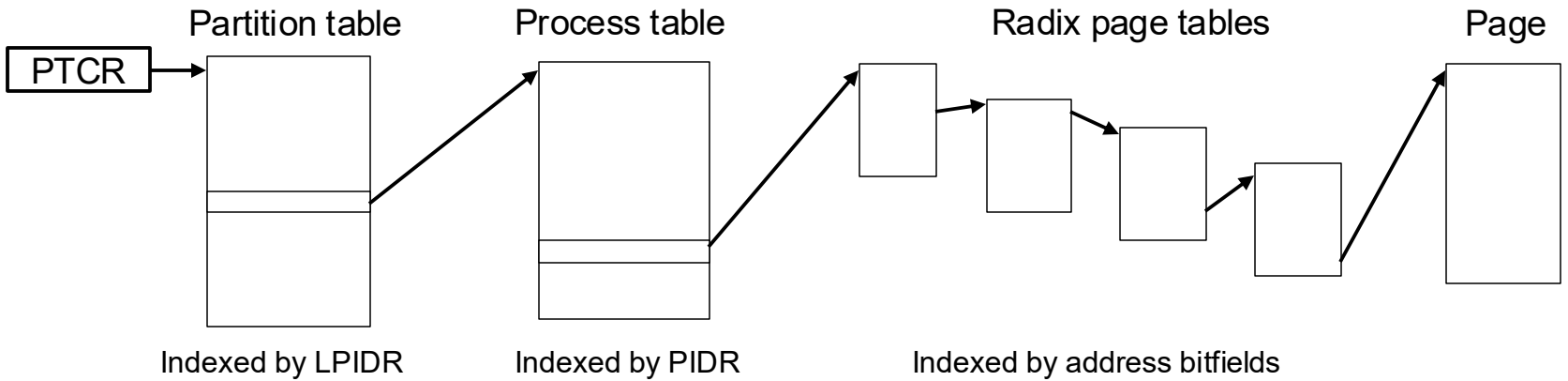
```
type PMUEventType is record
    no_instr_avail      : std_ulogic;
    dispatch            : std_ulogic;
    ext_interrupt       : std_ulogic;
    instr_complete      : std_ulogic;
    fp_complete         : std_ulogic;
    ld_complete         : std_ulogic;
    st_complete         : std_ulogic;
    br_taken_complete   : std_ulogic;
    br_mispredict       : std_ulogic;
    ipref_discard       : std_ulogic;
    itlb_miss           : std_ulogic;
    itlb_miss_resolved  : std_ulogic;
    icache_miss         : std_ulogic;
    dc_miss_resolved    : std_ulogic;
    dc_load_miss        : std_ulogic;
    dc_ld_miss_resolved : std_ulogic;
    dc_store_miss       : std_ulogic;
    dtlb_miss           : std_ulogic;
    dtlb_miss_resolved  : std_ulogic;
    ld_miss_nocache     : std_ulogic;
    ld_fill_nocache     : std_ulogic;
end record;
```

Memory management unit

- Level-1 TLBs in fetch1 and dcache
  - 4kB page size, looked up in parallel with cache tags
  - Instruction TLB is direct-mapped, 64 entries
  - Data TLB is 2-way set associative, 128 entries total
- MMU is a state machine
  - Sends a series of requests to the dcache to read the process table and PTEs
  - Eventually sends the translation to the dcache or fetch1
    - EA bits inserted if page size > 4kB
- Currently no caching of PTEs or PDEs in the MMU
  - Nor any caching of partial translation results (no “page walk cache”)
  - Just the L1 iTLB and dTLB inside the instruction and data caches
- Microwatt implements vestigial (1 entry) partition table

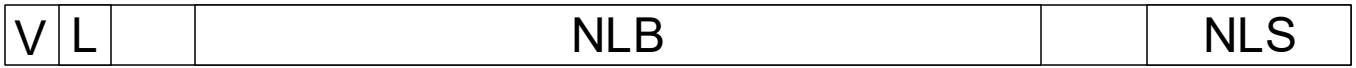
Memory management unit

- ISA defines two address translation schemes
  - Hashed page table (HPT), used by AIX and IBM/i and older Linux kernels
  - Radix page table, used by recent Linux kernels
- Microwatt implements radix and not HPT
- ISA defines a very general tree structure for radix trees



Memory management unit

- Radix tree page directory entry format:



NLS = Next level size, number of bits used to index next level of tree

- Two standard layouts defined in ISA
  - 52-bit EA, 4 levels, 64k page size (index fields 13, 9, 9, 5 bits)
  - 52-bit EA, 4 levels, 4k page size (index fields 13, 9, 9, 9 bits)
- Microwatt implements a general radix tree walker state machine
  - Any NLS value from 5 to 16 is permitted
  - Address space size from 31 bits (2 GiB) to 62 bits (4 EiB)
  - Any power-of-2 page size >= 4 kiB
  - Any number of levels between 1 and 10