

Improving Processor Performance

- Memory latency and performance
- Expanded register sets
- Instruction set
- Caches
- Pipelining



Making Computers Faster

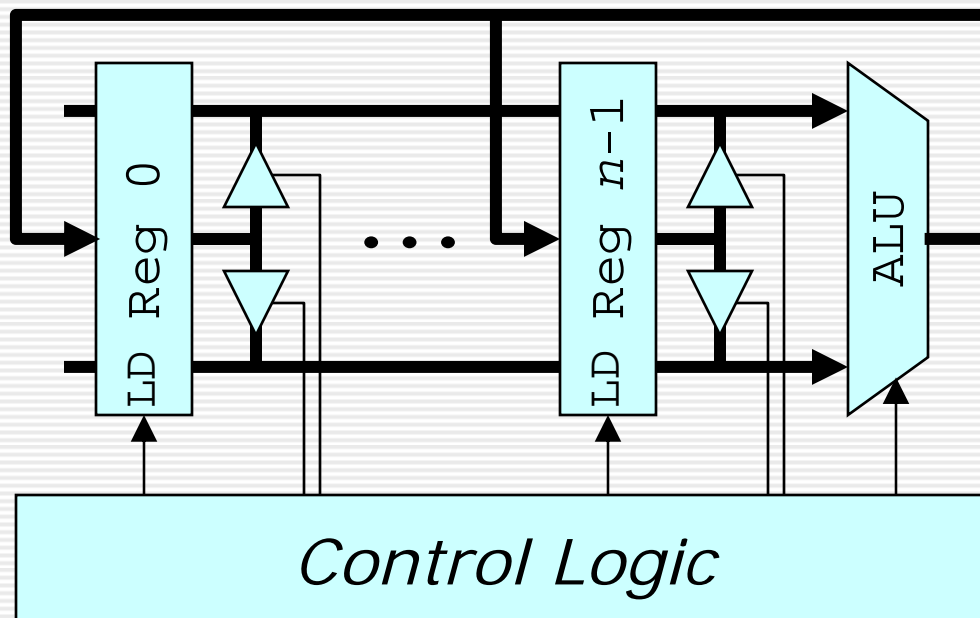
- Large & growing speed gap between CPU & DRAM.
 - » gate delays now less than 50 ps
 - » can take 100 ns to retrieve a word from DRAM
 - » get better performance by making fewer memory accesses
- Modern processors improve performance by
 - » using more extensive instruction sets
 - » providing additional addressing modes
 - » using multiple registers in place of single accumulator
 - » using small, fast cache memories to hold recently used instructions and data
 - » overlapping the execution of several instructions (pipelining)
 - example: fetch next instruction while executing current one
 - » providing multiple ALUs for parallel instruction execution

Extending Instruction Set

- Arithmetic and logic instructions
 - » integer add, subtract, multiply, divide
 - » word-wise AND, OR, NOT, EXOR, shift, rotate
 - » compare values (<, <=, =, >=, >)
 - » floating point add, subtract, multiply, divide, compare
- Conditional branch instructions
 - » sign of register or last operation performed
 - » result of comparison
 - » occurrence of arithmetic error
- Instruction coding must specify what registers to use for operands.
- Loads and stores may use register to specify address of memory location.

Processor with Multiple Registers

- Use of multiple registers can reduce number of memory accesses.
 - » modern processors have at least 32 general purpose registers
- Requires *Register File* and more general set of control signals.



Instruction Set for 16 Register CPU

- Instruction formats. Note, some require two words.

0	tdd	<i>Immediate Load.</i>	$R[t] \leftarrow dd.$ (sign-extended)
1	txx aaaa	<i>Direct Load.</i>	$R[t] \leftarrow M[aaaa].$
2	tsx	<i>Indexed Load.</i>	$R[t] \leftarrow M[R[s]+x].$
3	tsx	<i>Indexed Load with Increment.</i>	$R[t] \leftarrow M[R[s]+x]; R[s] \leftarrow R[s]+1.$
4	tsx	<i>Indexed Load with Decrement.</i>	$R[s] \leftarrow R[s]-1; R[t] \leftarrow M[R[s]+x].$
5	sxx aaaa	<i>Direct Store.</i>	$M[aaaa] \leftarrow R[s].$
6	tsx	<i>Indexed Store.</i>	$M[R[t]+x] \leftarrow R[s]$
7	tsx	<i>Indexed Store with Increment.</i>	$M[R[t]+x] \leftarrow R[s]; R[t] \leftarrow R[t]+1.$
8	tsx	<i>Indexed Store with Decrement.</i>	$R[t] \leftarrow R[t]-1; M[R[t]+x] \leftarrow R[s].$
9	0ts	<i>Copy.</i>	$R[t] \leftarrow R[s].$
9	1ts	<i>Add.</i>	$R[t] \leftarrow R[t]+R[s].$

92ts *Subtract.* $R[t] \leftarrow R[t] - R[s]$.
93ts *Negate.* $R[t] \leftarrow -R[s]$.
A0ts *And.* $R[t] \leftarrow R[t] \text{ and } R[s]$.
A1ts *Or.* $R[t] \leftarrow R[t] \text{ or } R[s]$.
A2ts *Exclusive-or.* $R[t] \leftarrow R[t] \text{ xor } R[s]$.
B000 *Halt.*
C0xx tttt *Branch.* $PC \leftarrow tttt$.
C1tt *Relative Branch.* $PC \leftarrow PC + tt$. (sign-extended
addition)
Dstt *Relative Branch on Zero.* **if** $R[s] = 0$ **then** $PC \leftarrow$
 $PC + tt$.
Estt *Relative Branch on Plus.* **if** $R[s] > 0$ **then** $PC \leftarrow$
 $PC + tt$.
Fstt *Relative Branch on Minus.* **if** $R[s] < 0$ **then** $PC \leftarrow$
 $PC + tt$.

```

entity cpu is port (
    clk, reset: in  STD_LOGIC;
    m_en, m_rw: out STD_LOGIC;
    aBus: out  STD_LOGIC_VECTOR(adrLength-1 downto 0);
    dBus: inout STD_LOGIC_VECTOR(wordSize-1 downto 0));
end cpu;
architecture cpuArch of cpu is
type state_type is ( reset_state, fetch, mload, dload, xload, ... );
signal state: state_type;
type tick_type is (t0, t1, t2, t3, t4, t5, t6, t7);
signal tick: tick_type;

signal pc: std_logic_vector(adrLength-1 downto 0); -- program counter
signal iReg: std_logic_vector(wordSize-1 downto 0); -- instr. reg.
signal maReg: std_logic_vector(wordSize-1 downto 0); -- mem. adr. reg.

type regFile is array(0 to 15) of std_logic_vector(wordSize-1 downto 0);
signal reg: regFile; -- register file

signal target: std_logic_vector(3 downto 0); -- target r
signal source: std_logic_vector(3 downto 0); -- source r

```

Same framework, but different states

RegFile replaces ACC, maReg stores base addr for load/store

```

begin
  process(clk) -- state transition process
  function nextTick(tick: tick_type) return tick_type is begin
    . . .
  end function nextTick;
  procedure decode is begin -- Instruction decoding.
    target <= ireg(11 downto 8); source <= ireg(7 downto 4);
    case iReg(15 downto 12) is
    when x"0" => state <= mload;
    when x"1" => state <= dload;
    when x"2" => state <= xload;
    . . .
    when x"9" => case ireg(11 downto 8) is
      when x"0" => state <= copy;
      when x"1" => state <= add;
      . . .
      when others => state <=halt;
    end case;
    target <= ireg(7 downto 4);
    source <= ireg(3 downto 0);
    . . .
  end procedure decode;
  procedure wrapup is begin . . . end procedure wrapup;

```

Store source and target in separate regs.

Extra decoding for arithmetic and logic inst.

```

begin
  if clk'event and clk = '1' then
    if reset = '1' then
      state <= reset_state; tick <= t0;
      pc <= (pc'range => '0'); iReg <= (iReg'range => '0');
      source <= (source'range => '0'); target <= (target'range => '0');
      maReg <= (maReg'range => '0');
      for i in 1 to 15 loop reg(i) <= (reg(i)'range => '0'); end loop;
    else
      tick <= nextTick(tick) ; -- advance time by default
      case state is
      when reset_state => state <= fetch; tick <= t0;
      when fetch =>
        case tick is
        when t1 => iReg <= dBus;
        when t2 => pc <= pc + '1';
          if ireg(15 downto 12) /= x"1" and
             ireg(15 downto 12) /= x"5" and
             ireg(15 downto 8) /= x"c0" then
            decode; tick <= t0;
          end if;
        when t4 => maReg <= dBus;
        when t5 => decode; pc <= pc + '1'; tick <= t0;
        when others => -- do nothing
        end case;
    end if;
  end if;
end

```

Quit early for
single word
instructions.

Load maReg
and proceed
to inst. exec.

```

-- load instructions
when mload =>
  if ireg(7) = '0' then -- sign extension
    reg(int(target)) <= x"00" & ireg(7 downto 0);
  else
    reg(int(target)) <= x"ff" & ireg(7 downto 0);
  end if;
  wrapup;
when dload =>
  if tick = t1 then reg(int(target)) <= dBus; end if;
  if tick = t2 then wrapup; end if;
. . .
-- register-to-register instructions
when copy =>
  reg(int(target)) <= reg(int(source));
  wrapup;
when add =>
  reg(int(target)) <=
    reg(int(target)) + reg(int(source));
  wrapup;
. . .
end process;

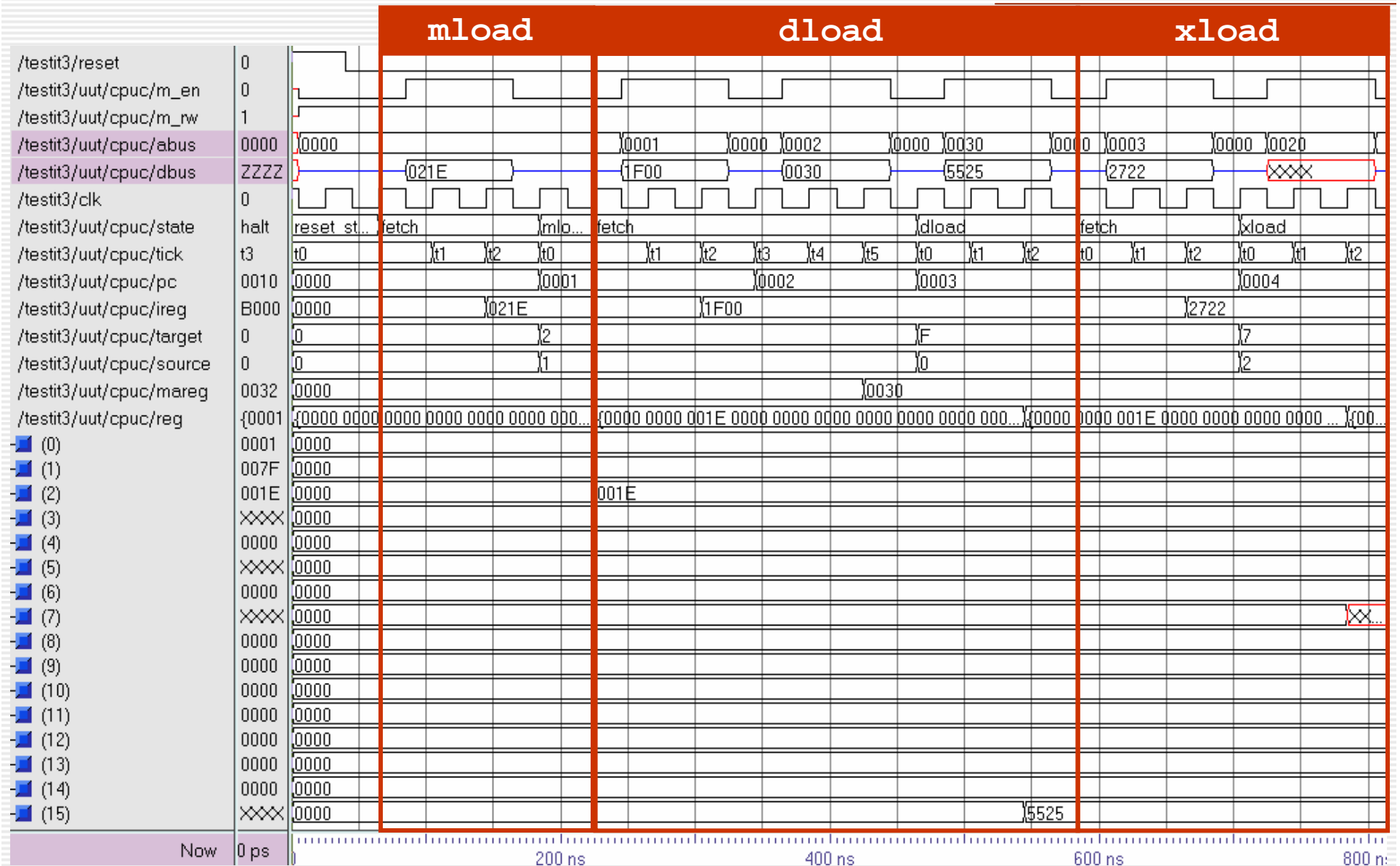
```

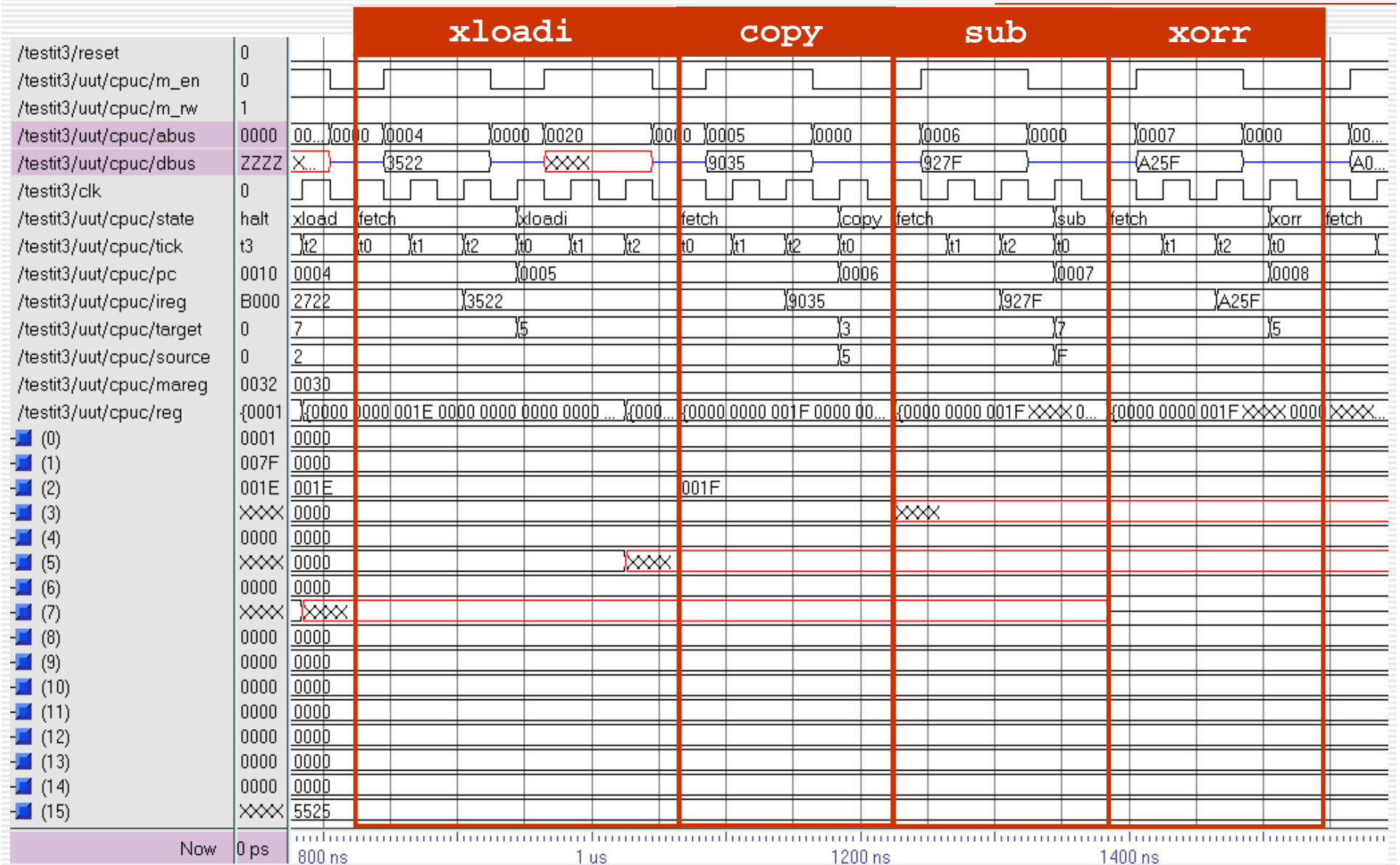
```

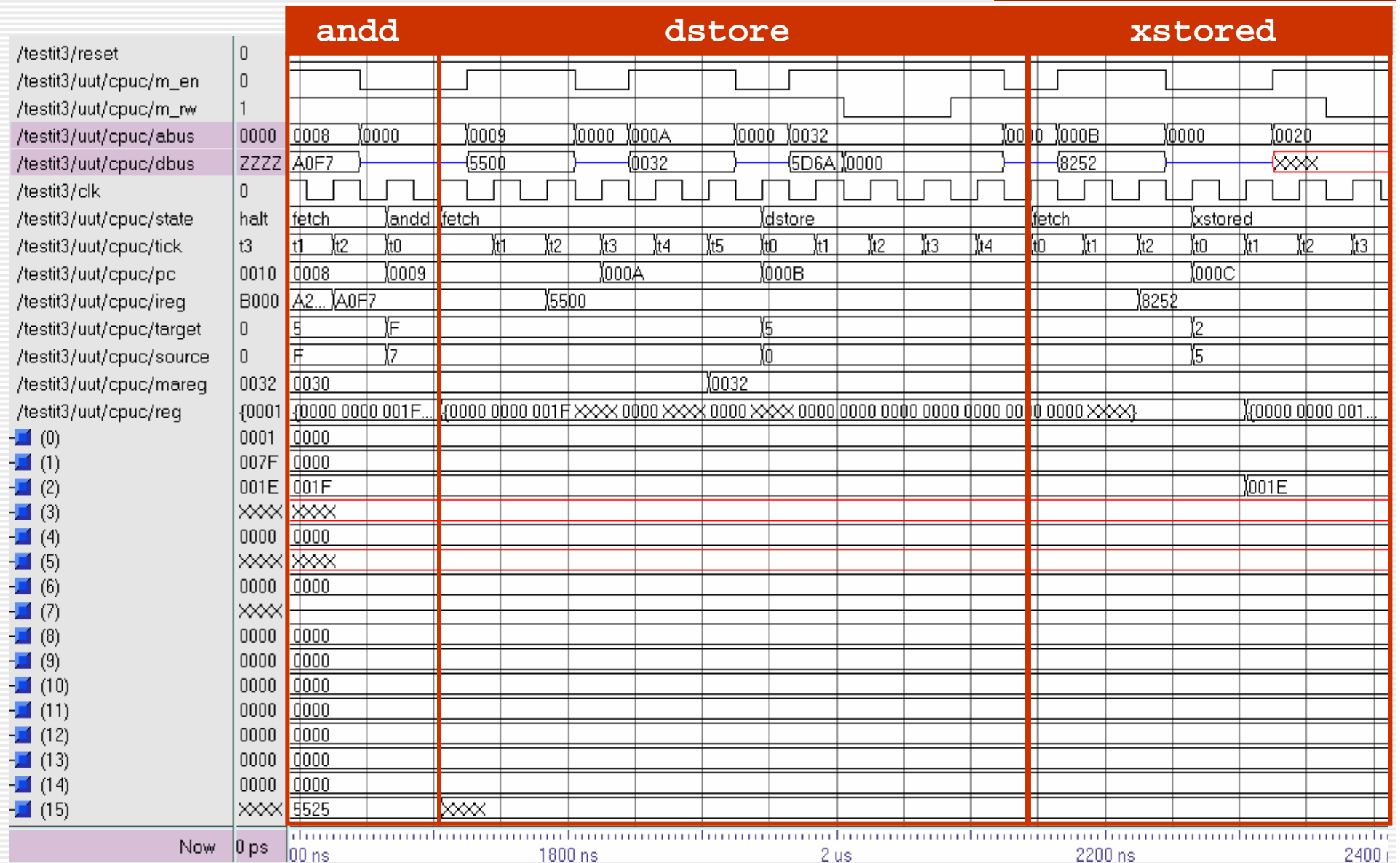
process(clk) begin -- perform actions that occur on falling clock edges
  if clk'event and clk = '0' then
    if reset = '1' then
      m_en <= '0'; m_rw <= '1';
      aBus <= (aBus'range => '0'); dBus <= (dBus'range => 'Z');
    else
      case state is
      when fetch =>
        if tick = t0 or tick = t3 then m_en <= '1'; aBus <= pc; end if;
        if tick = t2 or tick = t5 then
          m_en <= '0'; aBus <= (aBus'range => '0');
        end if;
        . . .
      when dstore =>
        if tick = t0 then m_en <= '1'; aBus <= maReg; end if;
        if tick = t1 then m_rw <= '0';
          dBus <= reg(int(source)); end if;
        if tick = t3 then m_rw <= '1'; end if;
        if tick = t4 then
          m_en <= '0'; aBus <= (abus'range => '0'); dBus <= (dBus'range => 'Z');
        end if;
        . . .
      end case;
    end if;
  end if;
end cpuArch;

```

Simulation Results

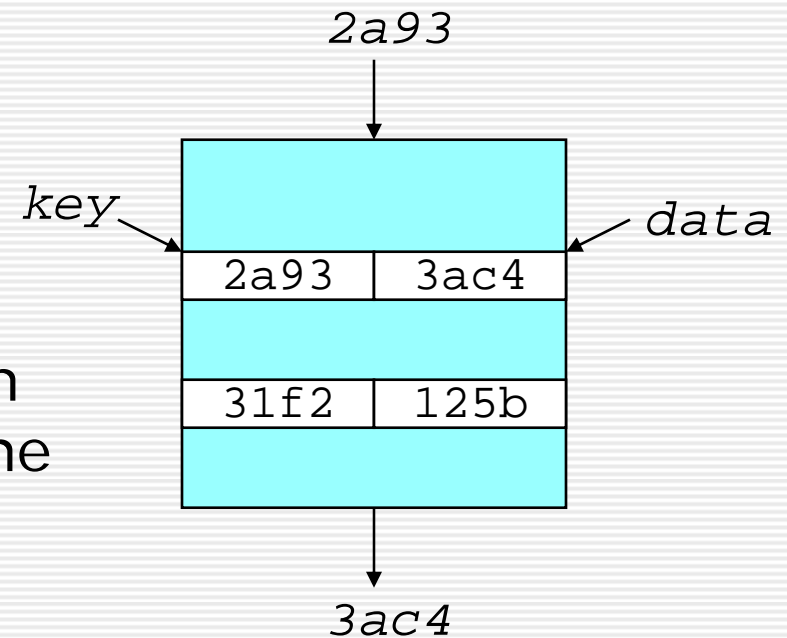






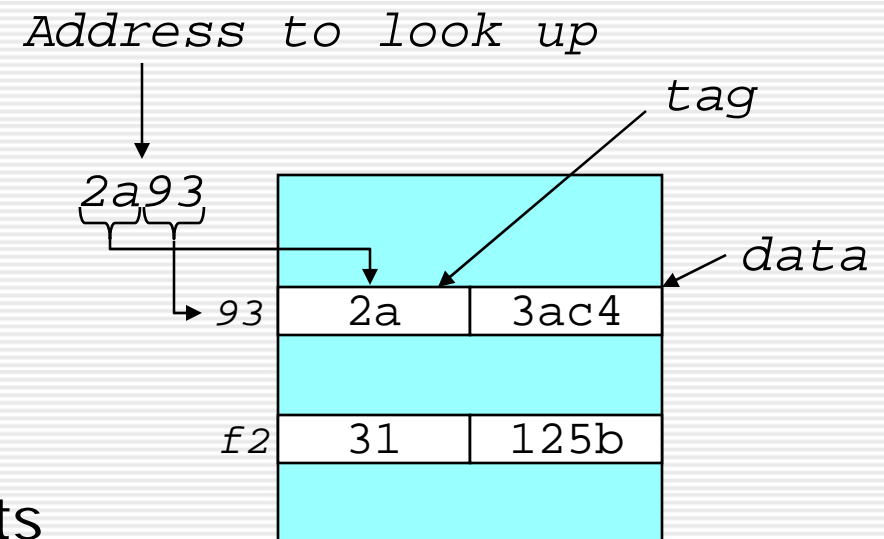
Fully Associative Caches

- A cache is a small memory that contains recently used words of memory.
- Conceptually simplest cache is the *fully associative cache* which stores (key, data) pairs.
 - » associative lookup using key to find data
 - » implementation involves parallel comparison of stored keys with "query key"
- In cache, main memory address used as key.
 - » before retrieving word from main memory, first check for it in cache
 - » retrieved words stored in cache



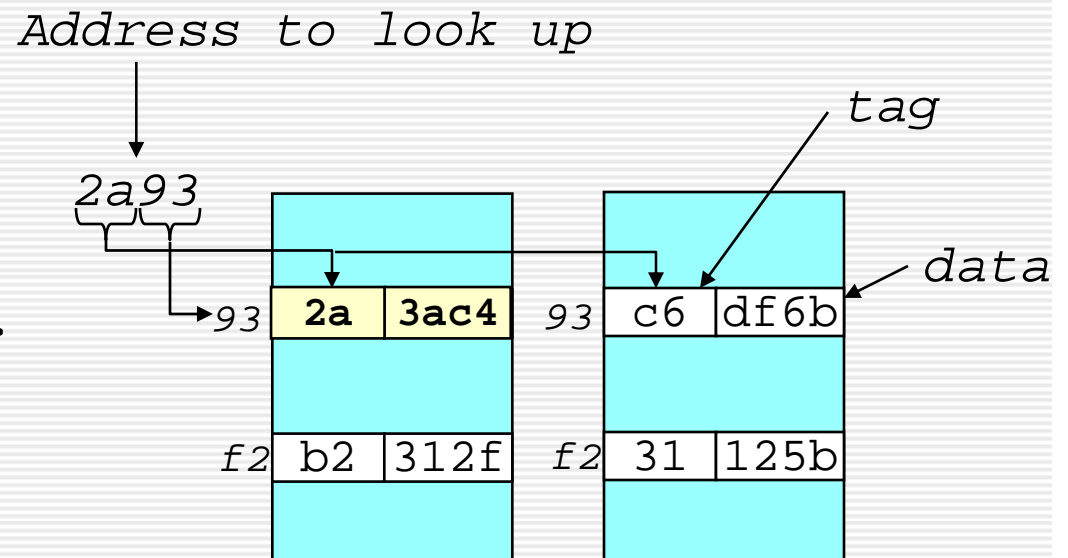
Direct-Mapped Caches

- Fully associative caches are too expensive to be cost-effective in most computers.
- A *direct mapped cache* is a less expensive alternative that uses SRAM and performs well in common cases.
 - » words stored at cache location specified by lower DRAM address bits
 - » higher DRAM address bits stored with data
 - » to see if DRAM word stored in cache, lookup using low bits and check tag against high bits
 - » works well for sequentially accessed DRAM locations



Set Associative Caches

- *Set associative caches* are an intermediate alternative between fully associative and direct-mapped caches.
 - » in a *2-way* s.a. cache, there are 2 SRAM banks and any given memory word can be stored in either one
 - » tags compared on lookup to see if either stored word matches address
- Better performance than direct-mapped.
- Less expensive than fully associative cache.
- Can be generalized to *N-way* (typically 4, 5).



More About Cache Operation

- Whenever a word is needed from memory, first check the cache and use the stored copy if possible.
- If word not in cache, fetch *cache line* containing required word and put it into cache (note delay).
 - » retrieved cache line replaces one of stored cache lines;
 - *replacement policy* determines which cache line is replaced
 - » for sequentially accessed data, fetching whole cache line speeds up later accesses
 - » updating memory determined by *write policy*
 - *write-through* – write to cache and memory together
 - *write-back* – write to memory when cache line replaced
- Many processors have multiple caches.
 - » *first-level cache* usually on-chip, separate instruction cache
 - » *second, third-level caches* are progressively larger, slower
- Cache consistency in multiple processor systems.

Pipelining

- Most modern processors use pipelining to improve performance.
- Simplest form overlaps instruction fetch, execution.
 - » if instructions are in instruction cache and data in data cache or registers, can nearly double effective processor speed
- By splitting instructions into several steps, parts of several instructions can be executed at same time.
 - » modern processors have as many as 20 pipeline stages
- Conditional branches hurt pipeline efficiency.
 - » *branch prediction hardware* attempts to guess which way branch will go, in order to keep pipeline busy
 - » quite effective for conditional branches in loops, which are very “predictable”