Why change our Multi-Cycle Design?

• Spending many cycles just moving data around
  – E.g. FETCH1: DrReg, LdA
• Spending significant % of each cycle on control
  – About 2.5 ns to generate control signals from IR
  – Sounds small, but this is >20% of each cycle
• Even “useful” cycles only using one unit
  – Either use ALU or read regs or read mem
• Small enhancements increase the waste
  – E.g. having “IncPC” saved one fetch cycle, and requires only a simple +4 adder
  – But this adder gets used in only once per instruction (one out of 5-8 cycles, depending on FSM and inst type)
Entire instruction done in only one cycle
• Data-moving takes only time it really needs (wires)
  – E.g. dedicated wire to take reg output to ALU input
• Control takes only as much time as it really needs
  – Figure out all control signals right after inst read from imem
  – Takes only part of one cycle, because one is all we have
• All major units get used in every cycle
  – Unless the unit is not needed for a specific instruction
• But…“enhancements” become “necessities”
  – Example: Branch uses ALU to compare, need separate PC+4+4*imm
  – Each cycle, need to read two registers read and write one
  – Etc.
• And… longest-to-do instruction determines clock
  – Usually a load, so all other insts will have slack for memory read
Supported: ADD
module RegFile(RADDR1,DOUT1,RADDR2,DOUT2,WADDR,DIN,WE,CLK);
  parameter DBITS; // Number of data bits
  parameter ABITS; // Number of address bits
  parameter WORDS = (1<<ABITS);
  parameter MFILE = ""
  reg [(DBITS-1):0] mem[(WORDS-1):0];
  input [(ABITS-1):0] RADDR1,RADDR2,WADDR;
  input [(DBITS-1):0] DIN;
  output wire [(DBITS-1):0] DOUT1,DOUT2;
  input CLK,WE;
  always @(posedge CLK)
    if(WE)
      mem[WADDR]=DIN;
  assign DOUT1=mem[RADDR1];
  assign DOUT2=mem[RADDR2];
endmodule

Three separate accesses
  • Two reads
  • One write
  Each with it own address!
wire [2:0] rregno1=rx, rregno2=ry;
wire [(DBITS-1):0] regout1, regout2;
// These come from decoding logic
// (reg becomes wire in non-edge always block)
reg [2:0] wregno;
reg wrreg;
reg [(DBITS-1):0] wregval;
// Now instantiate the register file module
RegFile #( .DBITS(DBITS), .ABITS(5), regFile(
    .RADDR1(rregno1), .DOUT1(regout1),
    .RADDR2(rregno2), .DOUT2(regout2),
    .WADDR(wregno), .DIN(wregval),
    .WE(wrreg), .CLK(clk)));
// ALU input 1 is always the first source register
wire [(DBITS-1):0] aluin1=regout1;
// ALU input 2 is decided by control logic (either regout2 or immediate)
reg [(DBITS-1):0] aluin2;
// Decided by control logic
reg [3:0] alufunc;
// Output of the ALU (becomes wire b/c of always block below)
reg [(DBITS-1):0] aluout;
always @(alufunc or aluin1 or aluin2)
  case(alufunc)
    OP2_AND:  aluout=aluin1&aluin2;
    ... // Same as in Project 2
    ... // but uses aluin1 and aluin2 instead of A and B
// Used by control logic for conditional branches
wire cond=aluout[0];
always @(opcode1 or opcode2 or ry or rz) begin
{aluimm, alufunc, isbranch, isjump, wrmem} =
{ 1'bX, 6'hXX, 1'b0, 1'b0, 1'b0};
{selaluout, selmemout, selpcplus, wregno, wrreg} =
{ 1'bX, 1'bX, 1'bX, 5'hXX, 1'b0};
case(opcode1)
    OP1_ALUR: {aluimm, alufunc, selaluout, selmemout, selpcplus, wregno, wrreg} =
               { 1'b0, opcode2, 1'b1, 1'b0, 1'b0, rz, 1'b1};
    OP1_ANDI: {aluimm, alufunc, selaluout, selmemout, selpcplus, wregno, wrreg} =
               { 1'b1, OP2_AND, 1'b1, 1'b0, 1'b0, ry, 1'b1};
...

Single-cycle

Supported: ADD, ADDI
Single-cycle

Supported:
All ALUR insts
Single-cycle

Supported:
All ALUR insts
LW
Single-cycle

Instr Mem

RF

Data Mem

Control

Supported:
All ALUR insts
LW, SW
Supported:
ADD
SUB, OR, etc.
ADDI, ANDI, etc.
LW
SW
BEQ, BNE
And it’s not done yet!

Supported:
ADD
SUB, OR, etc.
ADDI, ANDI, etc.
LW
SW
BEQ, BNE
JAL

Instr
Mem

RF

Control

ALU

Data
Mem

Supported:
ADD
SUB, OR, etc.
ADDI, ANDI, etc.
LW
SW
BEQ, BNE
JAL

Instr
Mem

RF

Control

ALU

Data
Mem

Supported:
ADD
SUB, OR, etc.
ADDI, ANDI, etc.
LW
SW
BEQ, BNE
JAL

Instr
Mem

RF

Control

ALU

Data
Mem

Supported:
ADD
SUB, OR, etc.
ADDI, ANDI, etc.
LW
SW
BEQ, BNE
JAL

Instr
Mem

RF

Control

ALU

Data
Mem
How do we do all this?

• Same approach we used before, except
• Units wired to each other, not to bus
  – Design becomes less modular
• The only real “state” is the PC, Regs, and Mem
  – The rest is just wires and combinatorial logic
• Central control takes opcode, generates all signals for the entire instruction!
• But can have other small pieces of control
  – E.g. decide if \( \text{PC}+4 \) or \( \text{PC}+4+\text{Offs} \) loaded into PC on a conditional branch instruction
Testing

• Same test-one-piece-at-a-time strategy
  – But all pieces are interconnected!
  – Can’t test some pieces without other pieces

• Test in order of instruction execution!
  – Write Verilog code to read instruction and inc PC
  – Display instruction (e.g. HEX) and PC (e.g. LEDR)
  – See if it works using SignalTap
  – Add registers and control for them, init registers with some values (.mif)
  – Display values being read, SignalTap to see if correct registers and values being output in each cycle
Testing

• Add ALU and control for it
  – See if ADD, SUB, AND, etc. work
• Add sign-extender and wire it up
  – Plus control to use it or reg value as appropriate
  – Test ADD, ADDI, etc.
• Add code for branch target address computation
  – Test if correct address computed for branches
• Add branch condition test, see if correct
• Wire them up - select PC+4 or PC+4+4*Offs
  – See if branches work
• Now add memory… you see where this is going
module MemArray(ADDR, DOUT, DIN, WE, CLK);

parameter DBITS; // Number of data bits
parameter ABITS; // Number of address bits
parameter WORDS = (1<<ABITS);
parameter MFILE = "";
input [(ABITS-1):0] ADDR;
input [(DBITS-1):0] DIN;
output reg [(DBITS-1):0] DOUT;
input CLK, WE;
(* ram_init_file = MFILE, ramstyle="no_rw_check" *)
reg [(DBITS-1):0] mem[(WORDS-1):0];
always @(posedge CLK)
  if(WE)
    mem[ADDR] <= DIN;
always @(ADDR)
  DOUT = mem[ADDR];
endmodule

• Form the ALU (not a register). We need a plan B!
• Addresses must come from FFs, not logic

• Problem:
  – We generate memory address using the ALU
  – Data address FF must get its value in the middle of the clock cycle!

• Hint:
  – Need a clock edge to trigger FF…
  – Needs to happen in the middle of a clock cycle…
• Memory address FF triggered by @(negedge clk)
• But clock frequency ends up being low (~25MHz)
  – PC -> mem -> alu -> address
takes much longer than just
  mem -> reg
  – But clock cycle must be set to allow
  PC -> address in only half a cycle
• Remember PLL settings?
  – Duty cycle: % of the cycle between pos and neg edge
  – Now at 50%, but can be changed to e.g. 60% or 70%
Duty Cycle

• We need
  – X: Delay from PC to memaddr
  – Y: Delay from memaddr to regin

• Add those up (X+Y)
  – This is the clock cycle time

• Compute what fraction of that is PC -> memaddr
  – X/(X+Y): This is the duty cycle

• Actually, need some slack for both X and Y
  – If posedge to negedge shorter than X, we’re in trouble
  – If negedge to posedge shorter than Y, trouble again
Faster! Compare to single-bus:

- Lower frequency and much more logic
  - Only 37.5MHz (with a 66% duty cycle), uses 64Kbits or RAM, 1139 reg bits, and 2345 logic
  - Compare to about 60MHz for single-bus multi-cycle, using 128Kbits or RAM, 1285 reg bits, and 1515 logic
- So single-cycle has 1.5X execution time...
- Not so! My stopwatch says:
  - 37.5MHz one-cycle design completes Sorter2.mif in 71s
  - 60MHz single-bus completes it in 205s!
  - So single-cycle is almost 3 times faster
  - Sweeet!
• Duty cycle trick only works for one thing
  – What if we need something like this twice per cycle?
  – There is only one negative edge
  – There are ways around this… we do have 4 PLLs
• Need a fancy PLL (maybe more than one)
• Real problem: very tricky to get right
  – What we are really doing is a multi-cycle processor
  – That has a fixed sequence of states
    • Cycle 1: Fetch, decode, do ALU
    • Cycle 2: Read mem and write result to register
  – And has different clock cycle times for each
We want a clean synchronous design!

• With “single-cycle” we have one long clock cycle
  – 26.7ns, with a 66% duty cycle (17.6ns then 9.1ns)
  – Would be 35.2ns if we can’t use the duty cycle trick

• With two-cycle: 17.6ns cycle, 35.2ns per inst
  – Cycle 1: fetch, decode, read reg, ALU
  – Cycle 2: mem, WrReg
  – Cycle 1 is what limits clock cycle to 17.6ns

• How do we split this into two cycles?
Insert FFs!

- Fetch, decode, read regs, ALU, decide PC
  - And make sure wrmem and wrreg are 0
- Latch everything we need in cycle 2 into FFs
  - Value from ALU, control signals
  - Value from regout2 (needed as data input to memory)
  - Value of pcplus (so JAL can write it to reg)
- Access memory, chose value, write register
  - And make sure the PC is not changed in this cycle!
    - BEQ/BNE completely done in cycle 1
    - JAL has its target address in cycle 1
    - And all other insts increment PC in cycle 1
What is the clock cycle time?

• We get Fmax of 45 MHz (22.5ns)
  – But we were supposed to get 17.6ns!
• What happened?
  – Latching into FFs takes time
    • Wire time to and from the FF can also be longer than direct wire time
  – But we had a FF in single-cycle too (memory address)
• So what happened?
  – Worst path was PC->iMem->Reg->ALU->dMemAddr!
    • This had to be done in 17.6ns (posedge to negedge of the one cycle)
  – But PC->iMem->Reg->ALU->cond->nextPC is longer!
    • Before it had to be done in one 26.7ns cycle, no problem!
    • Now it has to be done in one cycle, too, and 17.6ns not enough!
Option 1: Let longest path use second cycle
- End cycle 1 at the ALU output
- Compute next PC in cycle 2
- Now critical path is just PC->imem->reg->alu

Option 2: Split cycle 1 into multiple cycles
- Where? About half-way into the cycle…
- Where is that? Use TimeQuest to find out!
What is the half-way point?

• This is the longest path in my design
  – Use PC to read instruction memory: +3.6ns (3.8 total)
  – Read register file: +4.4ns (8.2 total)
  – Produce ALU result: +7.5ns (15.7 total)
    • This can be faster, we’ll look into it later
  – Use ALU result to decide if we should add regval1 or pcplus to imm (and then add): +6.5ns (22.2 total)
    • This is not very smart, we’ll look into it later
  – Get value to PC (iMem addr): +0.1ns (22.3 total)
  – The rest is clock delay, safety margin, etc.
So we split after reading registers...

• Stage 1:
  – Fetch, read regs
  – Do nothing in stages 2 and 3
    (no change in PC, no mem or reg write)

• Stage 2:
  – Do ALU operation, decide new PC
  – Get ALU output to dMem address FF
  – Do nothing in stage 3

• Stage 3:
  – Read/Write dMem, write regs
  – Do nothing in stage two (no PC change)
• Clock frequency is now 60MHz
  – Better than 45MHz that we had
• But each instruction takes one more cycle!
  – We had 45ns per instruction (2*22.5ns)
  – Now we have 51.1ns per inst (3*16.7ns)
  – It’s slower than what we had!
• Problem
  – We are decreasing clock cycle time
  – But we are increasing CPI even more
Pipelining

• Best of both worlds
  – Clock frequency increases with more stages
  – But we keep CPI at almost 1
• Caveat: figure out how to handle hazards, stalls, and forwarding
• We will build a basic pipeline with 3-5 stages
  – This is Project 3!
• So is it 3, 4, or 5 stages?
  – Depends on which gives us best performance!
  – And which you can get to work 😊