16 bit processor
A. Arhitectura generală: CPU

Modulul CPU interconectează unitățile funcționale printr-o arhitectură bazată pe o magistrală unică partajată (Single Bus Architecture).

Elemente de design:

1. System Bus (Magistrala de Sistem): Toate transferurile de date (Memorie - Regiștri - ALU) au loc pe wire [15:0] system_bus. Folosește logica Tri-state pentru a preveni conflictele:

- memory: Conduce magistrala când read_en e activ.
- registers: Conduce magistrala când reg_out_en e activ.
- ALU: Conduce magistrala când alu_out_en e activ.

2. Gestionarea Program Counter (PC): Deși există un registru PC în Register File, la nivelul CPU am implementat un registru dedicat PC pentru controlul fluxului. Acesta permite:

- Incrementare rapidă (PC + 1) în faza de FETCH.
- Încărcare directă (PC <= jump_addr) pentru instrucțiunile de salt (JMP, CALL).

3. Address Multiplexing: Adresa trimisă către memorie este selectată dinamic prin mem_addr_src:

- FETCH: Se folosește PC pentru a aduce instrucțiunea.
- LOAD/STORE: Se folosește adresa din instrucțiune (IR[9:0]).

4. MDR (Memory Data Register): Registru buffer MDR între memorie și ALU. Acesta capturează datele citite din memorie pentru a stabiliza semnalele înainte ca acestea să intre în unitatea aritmetică, asigurând stabilitatea datelor în timpul operațiilor multi-cycle.

5. Interfața ALU: Intrările ALU sunt selectate condiționat:

- Operand A: Vine din registrul X sau Y (selectat de bitul 8 al instrucțiunii).
- Operand B: Vine fie din memorie (via MDR), fie din câmpul imediat al instrucțiunii (via SignExtend), controlat de semnalul alu_src_b_imm.

B. Unitatea de Comanda: Control Unit

Control Unit-ul (CU) implementează o mașină cu stări finite (FSM) utilizând un registru de stare intern numit upc (Micro-Program Counter). Acesta realizează ciclul standard al procesorului: Fetch - Decode - Execute. Generează semnalele de control pentru magistrale, multiplexoare și semnalele de scriere/citire, asigurând că datele corecte ajung la locul potrivit la momentul potrivit.

Ciclul de Funcționare:

1. FETCH (Stările 0-1):
    - FETCH_1: Setează adresa memoriei la valoarea PC-ului.
    - FETCH_2: Așteaptă un ciclu pentru ca memoria sincronă să livreze datele, încarcă instrucțiunea în IR (Instruction Register) și incrementează PC-ul.

2. DECODE (Starea 2):
    - Analizează cei 6 biți de Opcode din instruction[15:10].
    - Tranzitează către starea specifică execuției instrucțiunii detectate (ex: starea 30 pentru ADD, starea 50 pentru JMP).
3. EXECUTE (Stările 10+):
    - Activează semnalele specifice (ex: alu_op, mem_write, reg_write_en).
    - Gestionarea Latenței Memoriei: Pentru instrucțiunile care accesează memoria (LOAD, Arithmetic cu operanzi din memorie), FSM-ul introduce stări de așteptare (ex: stările 60, 61) pentru a compensa latența de citire a RAM-ului sincron.
    - Operații Multi-Cycle: Pentru MUL, DIV și MOD, FSM-ul activează semnalul start și intră într-o buclă de *polling* (așteptare activă), monitorizând semnalele done. Execuția continuă doar după ce unitatea aritmetică semnalizează finalizarea calculului.

Controlul Fluxului: Instrucțiunile de salt (JMP, BRZ) controlează direct semnalele jump_en și jump_addr. În cazul BRZ (Branch if Zero), saltul este activat condiționat, doar dacă bitul corespunzător din registrul de flag-uri (flags[3]) este setat.

1. Sincronizarea cu Memoria: "Un aspect critic al designului este gestionarea memoriei sincrone. Deoarece citirea din memorie durează un ciclu de ceas, am introdus stări intermediare (Wait States) în automatul de control. Dacă aș fi încercat să citesc și să folosesc data în același ciclu, aș fi citit date vechi sau 'garbage'."
2. Control Unit-ul este proiectat să suporte operații care durează un timp variabil. Pentru înmulțire și împărțire, unitatea nu presupune că rezultatul e gata imediat. Ea trimite un semnal de start și așteaptă un semnal de done.
3. Micro-codare: "Folosirea structurii case (upc) face codul foarte ușor de depanat și extins. Fiecare instrucțiune are propria 'micro-secvență' clar definită."
4. Decuplarea ALU: "Control Unit-ul decide *sursa* operanzilor ALU (fie din memorie, fie o valoare imediată din instrucțiune) prin semnalul alu_src_b_imm, oferind flexibilitate maximă setului de instrucțiuni.”

C. Unitatea Aritmetică și Logică: ALU

Modulul ALU (Arithmetic Logic Unit) centralizează toate operațiile matematice și logice ale procesorului. Primește doi operanzi pe 16 biți (A și B), un cod de operație (op) și returnează rezultatul calculului împreună cu starea flag-urilor (Zero, Negative, Carry, Overflow).

Descrierea Unității Aritmetice și Logice (ALU) ALU este implementat folosind o arhitectură structurală ierarhică. Modulul de top acționează ca un wrapper și un multiplexor.

Organizare Internă:

1. Instanțiere Paralelă: Toate unitățile funcționale (Adders, Shifters, Logic Gates, Multipliers) sunt instanțiate fizic în paralel.
    - Operațiile combinaționale (ex: And, parallel_adder) calculează rezultatul continuu pe baza intrărilor A și B.

    - Operațiile secvențiale (ex: div, multip) sunt activate doar la primirea semnalului start și semnalizează finalizarea prin done.

2. Selecția Rezultatului (Multiplexare): Un bloc always @(*) funcționează ca un multiplexor mare controlat de intrarea op (5 biți). Acesta selectează:

    - Rezultatul corect (result) din ieșirea sub-modulului corespunzător.
    - Setul de flag-uri asociat operației (Z, N, C, V).

Detalii Specifice:

- Înmulțirea (MUL): Returnează un rezultat pe 32 de biți, împărțit în result (partea Low) și result_high (partea High), esențial pentru algoritmi criptografici sau calcule de precizie.
- Comparație și Testare (CMP, TST): Aceste instrucțiuni (Opcode 15 și 17) actualizează doar flag-urile, fără a modifica registrul de date (result este ignorat de Control Unit în aceste cazuri, dar flag-urile sunt salvate).
- Calculul Flag-urilor: Fiecare operație are logică dedicată pentru determinarea flag-urilor. De exemplu, o scădere (SUB) poate genera un flag de Overflow (V) diferit față de o deplasare (LSL), iar ALU asigură că procesorul vede doar flag-urile relevante pentru operația curentă.
- ALU integrează unități hardware pentru Adunare, Scădere, Înmulțire, Împărțire și Modulo. Acestea nu sunt simulate software, ci sunt blocuri hardware dedicate care rulează independent, declanșate de Control Unit.

D. Resurse de stocare: memory and registers

Memoria

Acesta este un bloc de memorie RAM sincronă (Synchronous RAM), care servește drept Memorie de Instrucțiuni și Date. Arhitectura permite ca atât codul cât și datele sa împartă același spațiu de adresare.

Modulul memory implementează o memorie volatilă de 1024 x 16 biți. Accesul la memorie este controlat printr-o magistrală de adrese pe 10 biți (pentru a acomoda formatul instrucțiunilor de JMP și CALL, care alocă exact 10 biți pentru adresa țintă) și o magistrală de date bidirecțională (in/out) pe 16 biți.

Organizare Internă:

- Capacitate: 1024 locații (adrese 0-1023), fiecare stocând un cuvânt de 16 biți.
- Magistrala in/out: Portul data este bidirecțional. Folosește logică tri-state (Z) pentru a permite atât scrierea cât și citirea pe același set de fire, economisind resurse de rutare (simplifică interconectarea cu procesorul, reducând numărul de fire necesare pe chip, deoarece aceleași linii sunt folosite și pentru a citi instrucțiuni, și pentru a salva rezultate).
- Memorie Unificată: Această memorie găzduiește: codul mașină, variabilele globale și Stiva (Stack-ul).

Temporizare și Control:

- Sincronizare: Toate operațiile de actualizare internă au loc pe frontul pozitiv al ceasului (posedge clk).

- Scriere (Write): Când write_en este activ, valoarea de pe magistrala data este scrisă la adresa addr.
- Citire (Read): Citirea are o latență de 1 ciclu de ceas deoarece este sincronă. Procesorul trebuie să țină cont de faptul că datele cerute la ciclul T vor fi disponibile pe magistrală în ciclul T+1.
    1. La posedge clk, dacă nu se scrie, datele de la addr sunt încărcate în registrul intern read_data.
    2. Logic, dacă read_en este activ (și write_en inactiv), buffer-ul tri-state se deschide și pune valoarea read_data pe magistrala externă data.

Inițializare: Modulul include un bloc initial care pune pe zero întregul spațiu de memorie la pornirea simulării, asigurând un comportament predictibil înainte de încărcarea oricărui program.

Registrii

Rolul modulului în arhitectură

Acest modul stochează starea internă a procesorului. El conține toți regiștrii specificați în cerințe și permite scrierea (sincronă) și citirea (asincronă/combinatorie) a datelor pe magistrala comună, precum și expunerea directă a valorilor către alte unități (precum ALU).

Regiștri Generali și Acumulator:

- reg [15:0] acc_reg corespunde cerinței "0 16-bit Accumulator”.r
- reg [15:0] x_reg, y_reg corespund cerinței "2 16-bit general purpose (GP) registers: X and Y”.

Regiștrii de Control:

- reg [3:0] fr_reg corespunde cerinței "4-bit Flag register: Zero, Negative, Carry, Overflow”.
- reg [15:0] sp_reg corespunde cerinței "16-bit stack pointer”.
- reg [15:0] pc_reg corespunde cerinței "Program Counter".

Modulul registers gestionează cei 6 regiștri esențiali ai arhitecturii pe 16 biți. Accesul la regiștri se face prin intermediul a două porturi de selecție pe 3 biți: s_in (pentru scriere) și s_out (pentru citire pe magistrală).

Mecanismul de Adresare: Regiștrii sunt mapați intern astfel:

- 000: ACC (Accumulator)
- 001: X (General Purpose)
- 010: Y (General Purpose)
- 011: FR (Flags - 4 biți)
- 100: SP (Stack Pointer)
- 101: PC (Program Counter)

Logica de Scriere: Scrierea este sincronă pe frontul pozitiv al ceasului (posedge clk) și este activată de semnalul write_en. În cazul registrului de Flag-uri (fr_reg), deși magistrala de date d_in este de 16 biți, se rețin doar cei 4 biți nesemnificativi (LSB).

Logica de Citire: Citirea pe portul d_out este controlată de semnalul out_en. Modulul implementează logica tri-state (16'bz când out_en este 0), permițând conectarea directă la o magistrală de date partajată fără conflicte electrice. La citirea registrului de Flag-uri (4 biți), valoarea este extinsă cu zerouri la 16 biți ({12'b0, fr_reg}).

Ieșiri Dedicate: Pe lângă ieșirea multiplexată d_out, modulul expune valorile tuturor regiștrilor prin porturi dedicate (ex: acc_out, pc_out). Acest lucru permite unităților precum ALU sau Control Unit să citească starea curentă a regiștrilor (ex: PC pentru incrementare, Flag-uri pentru Branch) independent de traficul de pe magistrala de date.

E. Setul de instrucțiuni: assembler

Acest script este un utilitar software de tip Two-Pass Assembler. Rolul său este de a traduce codul sursă scris în limbaj de asamblare (mnemonice precum ADD, JMP, etichete) în cod mașină binar pe 16 biți, formatat specific pentru a fi încărcat în memoria procesorului tău.

- Formatul Instrucțiunilor:
    - Format 1 (ALU/Load/Store): Opcode (6 biți) + Reg (1 bit) + Immediate (9 biți).
    - Format 2 (Branch/Jump): Opcode (6 biți) + Address (10 biți).
- Testarea core-ului se face prin implementarea unor aplicații traduse în cod mașină folosind un assembler.

Pentru a eficientiza scrierea programelor de test, am dezvoltat un asamblor în Python care automatizează traducerea codului. Acesta funcționează în doi pași (Two-Pass Assembly):

Pasul 1 (Symbol Resolution):

- Parcurge fișierul sursă linie cu linie.
- Elimină comentariile (după ;) și spațiile inutile.
- Identifică etichetele (ex: LOOP:) și le asociază cu adresa curentă a instrucțiunii în memoria programului. Acestea sunt salvate într-un Symbol Table. Această etapă este critică pentru a permite salturi "înainte" (forward references) în cod.

Pasul 2 (Code Generation):

- Reparcurge codul și traduce fiecare instrucțiune.
- Mapping: Înlocuiește mnemonicele (ex: ADD) cu codurile binare corespunzătoare (ex: 000011).
- Operanzi:
    - Pentru instrucțiunile de tip ALU, convertește valorile imediate în binar (inclusiv numere negative folosind Complement față de 2).
    - Pentru instrucțiunile de salt (JMP, BRZ), înlocuiește etichetele cu adresele numerice calculate în Pasul 1.

Output: Generează două fișiere:

1. .txt: Un fișier text cu șiruri binare (ex: 0000110000000101), ideal pentru funcția $readmemb din Verilog Testbench.

2. .bin: Un fișier binar real, util dacă s-ar dori scrierea pe un ROM fizic.

F. Aplicație: Pocket Calculator

Acesta este un program scris în C care folosește biblioteca ncurses pentru a crea o interfață grafică în consolă. Rolul său este de a acționa ca un Wrapper de Sistem:

1. Preia input de la tastatură (numere, operații).
2. Scrie aceste date într-un fișier intermediar (input.txt).
3. Lansează automat simularea Verilog (system("./run_sim.sh")).
4. Citește rezultatul calculat de procesor din fișierul de ieșire (result.txt).
5. Afișează rezultatul utilizatorului.

Comunicarea dintre interfața C și procesorul simulat în Verilog se face prin fișiere partajate, simulând o memorie mapată:

1. Generarea Input-ului: Funcția run_simulation primește operanzii A, B și codul operației. Aceasta creează fișierul input.txt formatat hexazecimal, care va fi încărcat în memoria procesorului de către Testbench.

2. Execuția Hardware: Programul C apelează system("./run_sim.sh"). Acest script execută iverilogși vvp (Icarus Verilog), rulând logica hardware definită în CPU.v și ALU.v. Procesorul execută instrucțiunile, calculează rezultatul și îl scrie în memoria sa, care este apoi dump-uită în result.txt.

3. Preluarea Rezultatului: Imediat ce simularea se încheie, programul C citește result.txt și actualizează ecranul.

Interfața Grafică (Ncurses): Utilizează o buclă infinită (while(1)) pentru a scana tastele. Gestionează o mașină de stări simplă (state = 0 pentru Input A, 1 pentru Input B, 2 pentru Rezultat) pentru a oferi o experiență de utilizare fluidă, similară unui calculator fizic.