به نام خدا

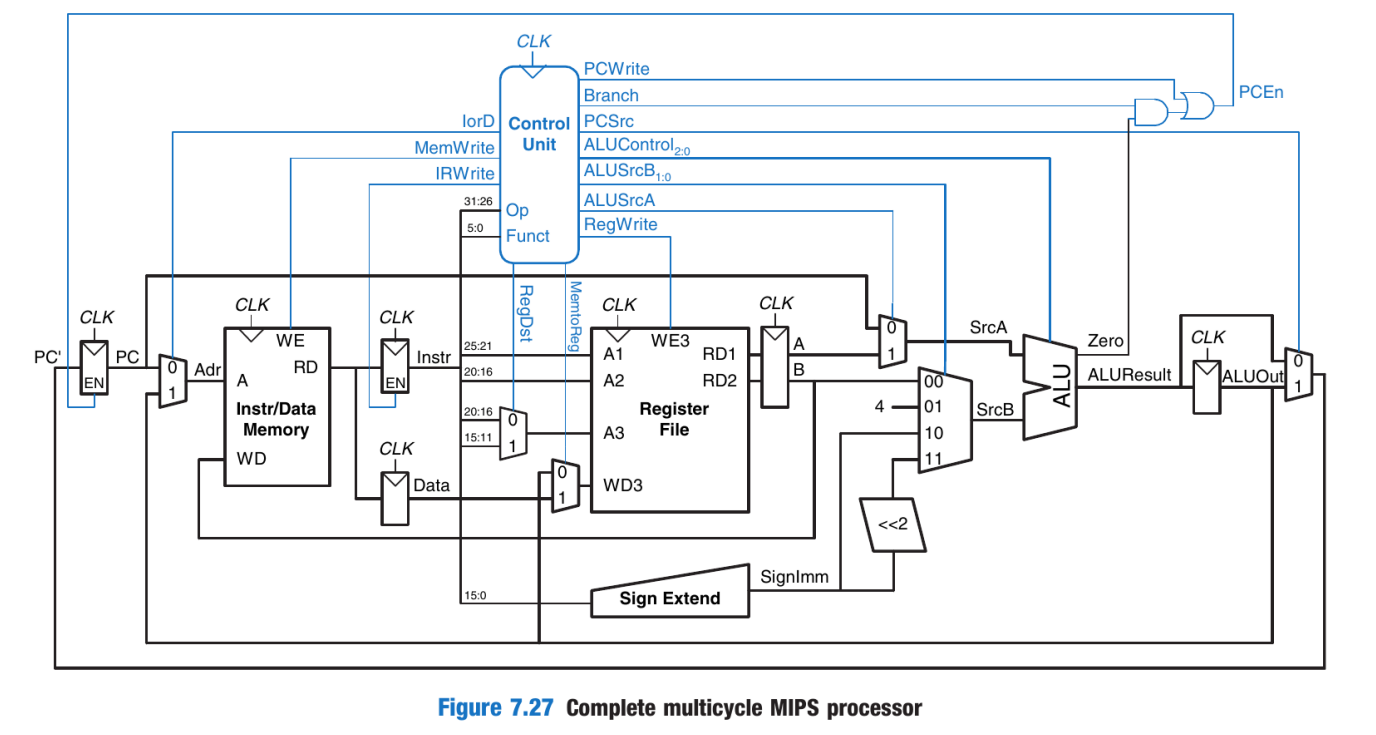
گزارشکار پروژه طراحی کامپیوتری سیستم‌های دیجیتال

اعضای گروه: نگین صمصامی ، رسول باقری

**موضوع پروژه: طراحی و پیاده‌سازی پردازنده (Multi-Cycle) MIPS**

پیاده‌سازی کامل مسیر داده (Data Path) و واحد کنترل (Control Unit) یک پردازنده MIPS چندمرحله‌ای به همراه شبیه‌سازی و تست سخت‌افزارهای داخلی شامل رجیسترها، مالتی‌پلکسرها، واحد ALU و حافظه، با استفاده از زبان توصیف سخت‌افزار Verilog و تحلیل عملکرد در نرم‌افزار شبیه‌ساز

**Data Path :**



در معماریmulticycle ، هر دستور در چند **سیکل کلاک** اجرا می‌شود و سخت‌افزارهای مشترک در سیکل‌های مختلف **بازاستفاده** می‌شوند. برای این کار، مسیر داده دارای چند رجیستر موقتی و مالتی‌پلکسر است تا مقادیر بین مراحل حفظ و انتخاب شوند. اجزای اصلی:

**اجزای کلیدی:**

* **: PC (Program Counter)** آدرس دستور فعلی با سیگنال **PCEn** به‌روزرسانی می‌شود.
* **:** **Memory (Instr/Data)** حافظه مشترک دستور/داده.  
  کنترل‌ها) **IorD** : انتخاب آدرسPC : یا ALUOut )**MemWrite ,** (نوشتن( )**IRWrite ,**بارگذاری دستور(
* **: IR (Instruction Register)** نگهداری دستور خوانده‌شده از حافظه.
* **: MDR (Memory Data Register)** نگهداری داده خوانده‌شده از حافظه تا سیکل بعد.
* **: Register File**  رجیسترهای معماری.  
  کنترل‌ها ) **RegDst** :انتخاب مقصد (rt/rd : **MemtoReg** , )منبع نوشتن :, (ALUOut/MDR **RegWrite**
* **A و : B** رجیسترهای موقتی خروجی خواندن رجیستر فایل ( ورودی‌های ALU را یک سیکل نگه می‌دارند).
* **Sign Extend و : ShiftLeft2** تولید SignImm و نسخه‌ی شیفت‌خورده‌ی آن برای محاسبات شاخه/آدرس مؤثر.
* **MUX ها:**
  + **: ALUSrcA**  انتخاب ورودی A‌از ALU بین PC و رجیستر A
  + **: ALUSrcB[1:0]** انتخاب ورودی B‌از ALU بین , B ثابت SignImm , 4 و SignImm<<2
  + **: IorD** انتخاب منبع آدرس حافظه ( PC یا ALUOut)
  + **: RegDst / MemtoReg** انتخاب رجیستر مقصد و داده‌ی نوشتن در رجیستر فایل.
  + **: PCSrc**  انتخاب منبع ورودی ) PC بین ALUResult و (ALUOut
* **: ALU** انجام عملیات منطقی/حسابی با کنترل **ALUControl[2:0]** و تولید **ALUResult** و **Zero** .
* **: ALUOut** رجیستر نگهدارنده‌ی خروجی ALU برای استفاده در سیکل‌های بعد)مثل آدرس مؤثر یا نتیجه(R-type .
* **: PCEn**از **OR (PCWrite) (Branch & Zero)** تولید می‌شود تا فقط در زمان‌های مجاز PC به‌روزرسانی شود.

# **جریان داده در مراحل (چرخه‌های اجرای دستور)**

### Instruction Fetch (IF)-1

* → **IorD=0** آدرس حافظه PC = .
* → **IRWrite=1** دستور از حافظه به **IR** می‌رود.
* **ALUSrcA=0** **(** انتخاب **PC** ) **و ) ALUSrcB=01**ثابت **ALU**→ **(** 4 مقدار PC+4 را می‌سازد.
* **PCSrc=0** و → **PCWrite=1** با فعال شدن **PCEn ,** PC ← PC+4 .

### Instruction Decode / Register Fetch (ID) -2

* خواندن RS و RT از **Register File** و ذخیره در **A** و **B**.
* **SignExtend** تولید SignImm می‌کند و SignImm<<2 نیز آماده می‌شود.
* هم‌زمان، برای آدرس شاخه **ALUSrcA=0** (PC) و **ALUSrcB=11** (SignImm<<2) → **ALUResult** شاخه‌ی احتمالی را می‌سازد و در /**ALUOut**یا **ALUResult** نگه داشته می‌شود .

### 3- Execute / Address Calculation (EX)

* **R-type: ALUSrcB=00** (B) , **ALUSrcA=1** (A) , **ALUControl** از → funct نتیجه در **ALUOut.**
* **Load/Store**: **ALUSrcA=1 , ALUSrcB=10** (SignImm)→ ALUOut = A + SignImm )آدرس مؤثر(.
* **Branch (BEQ)**: **ALUSrcA=1 , ALUSrcB=00** → ALU تفریق می‌کند؛ اگر **Zero=1**و **Branch=1**آنگاه **PCEn**فعال و **PCSrc**منبع شاخه را انتخاب می‌کند (معمولاً ALUOutکه در ID محاسبه شده(

### Memory Access (MEM) -4

* **SW**: **IorD=1** (آدرس ازALUOut) **MemWrite=1** → نوشتن مقدار **B** در حافظه.
* → **LW**: **IorD=1** خواندن از حافظه و ذخیره در **MDR .**

### Write Back (WB)-5

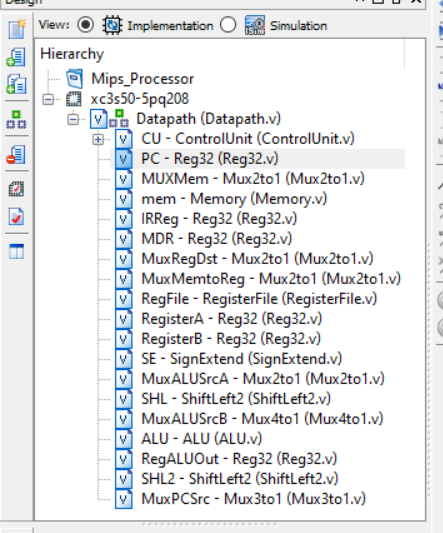
* **R-type**: **RegDst=1** (rd) **, MemtoReg=0** )از ( ALUOut و **RegWrite=1.**
* **LW**: **RegDst=0** (rt) , **MemtoReg=1** )از (MDR و **RegWrite=1.**

 با چند سیکل کردن اجرا، یک **ALU** و یک **Memory** برای همه مراحل مشترک می‌مانند ⇒ **هزینه سخت‌افزار کمتر**.

 رجیسترهای میانی (**IR, MDR, A, B, ALUOut**) داده‌ها را بین سیکل‌ها نگه می‌دارند تا بازاستفاده‌ی منابع ممکن شود.

 سیگنال‌های کنترلی از **Control Unit (FSM)** می‌آیند و هر سیکل مسیر داده را برای همان مرحله تنظیم می‌کنند.

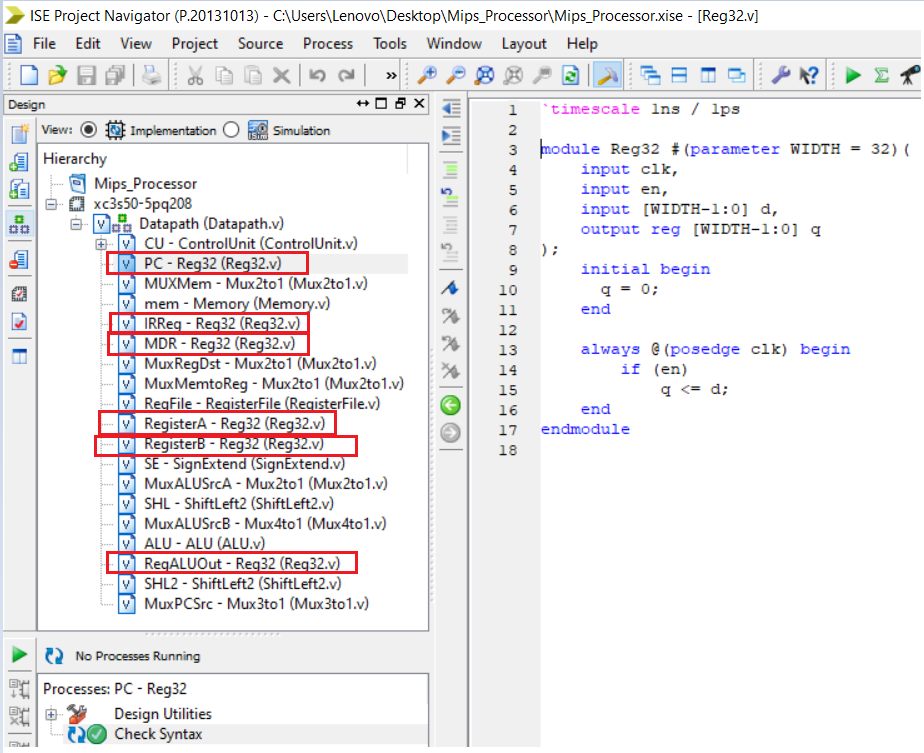
**شمای کلی از ماژول های استفاده شده در datapath**



**تصاویر و توضیحات ماژول های استفاده شده در datapath :**

### ****نام ماژول Reg32:****

**نوع: ثبات (Register) با ورودی Enable**



این ماژول در معماری **پردازنده MIPS MultiCycle**  نقش یک **Register عمومی ۳۲ بیتی را دارد.**

### ****عملکرد کلی:****

* این ماژول یک رجیستر لبه‌بالارونده (posedge clock) است.
* فقط زمانی که سیگنال en برابر 1 باشد، داده‌ی ورودی d در رجیستر ذخیره می‌شود.
* در حالت اولیه (initial) مقدار خروجی q صفر می‌شود تا پردازنده با حالت مشخص شروع کند.

### ****ورودی‌ها و خروجی‌ها:****

* :clk کلاک – هر تغییر روی لبه‌ی بالارونده اعمال می‌شود.
* :en فعال‌ساز (Enable) – تعیین می‌کند آیا رجیستر مقدار جدید بگیرد یا نه.
* :d داده ورودی – مقداری که باید در رجیستر ذخیره شود.
* :q داده خروجی – مقدار فعلی ذخیره‌شده در رجیستر.

### ****نقش در Data Path:****

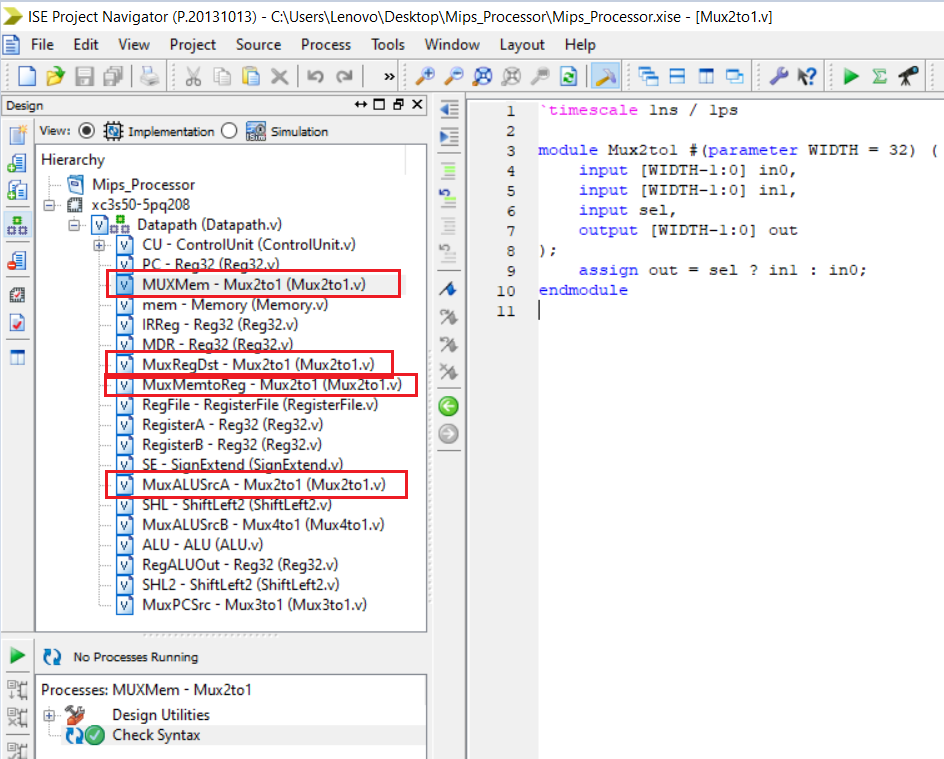
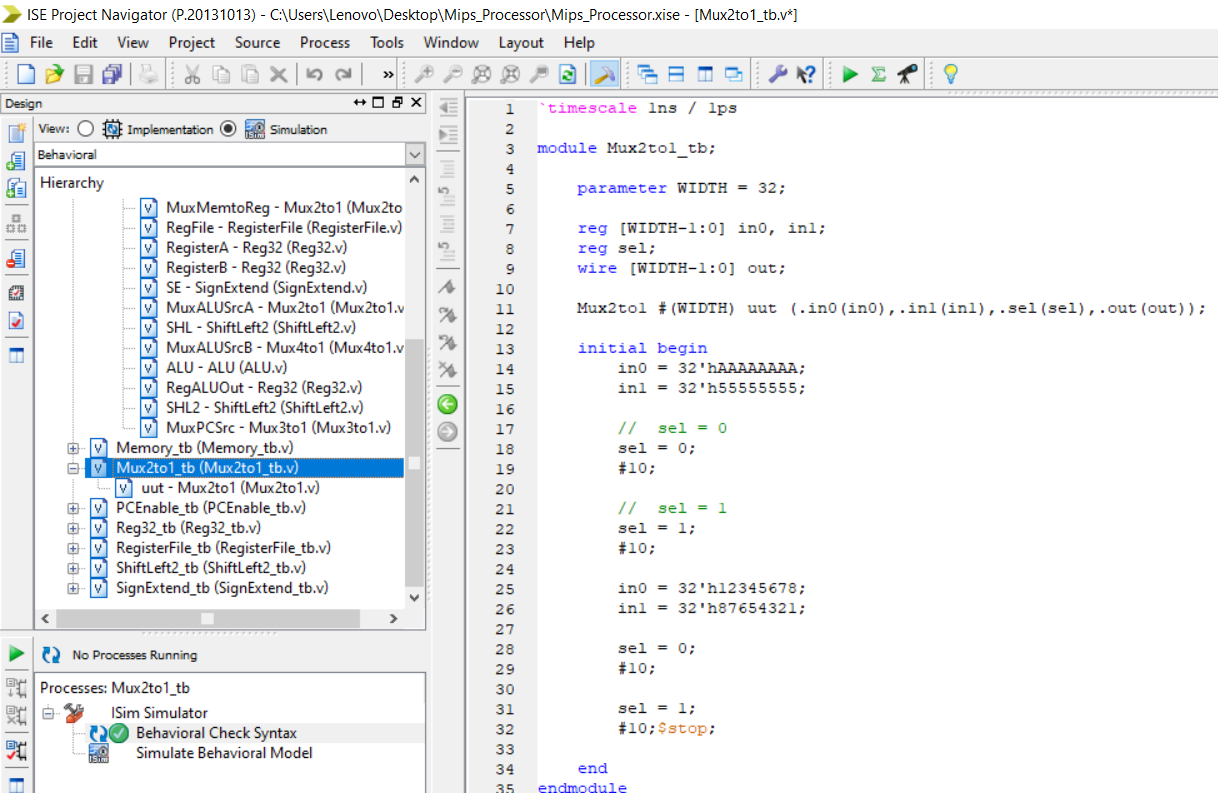
* چنین رجیستری می‌تواند برای **IR (Instruction Register)**، **MDR (Memory Data Register)**، **A/B Register** (خروجی فایل رجیسترها) یا **ALUOut** به‌کار رود.
* چون این رجیستر **پارامتریک** است، می‌شود در هر نقطه از مسیر داده که به رجیستر با پهنای خاص نیاز داریم استفاده شود.
* سیگنال en توسط **واحد کنترل** تعیین می‌شود تا در سیکل مناسب مقدار رجیستر به‌روز شود.

### ****ارتباط با مراحل اجرای دستور در MIPS چند سیکلی:****

* :**IF (Instruction Fetch)** رجیستر IR مقدار جدید دستور را از حافظه می‌گیرد.
* : **ID (Instruction Decode)** رجیسترهای A و B داده خوانده‌شده از فایل رجیسترها را ذخیره می‌کنند.
* :**EX/MEM/WB** رجیستر ALUOut و MDR برای نگه‌داری نتایج موقت استفاده می‌شوند.

### ****نام ماژول Mux2to1:****

**نوع: مالتی‌پلکسر دو به یک با پهنای قابل تنظیم**



این ماژول یک **مالتی‌پلکسر دو به یک (2-to-1 MUX)** پارامتریک است که در طراحی پردازنده MIPS MultiCycle برای انتخاب بین دو ورودی مختلف استفاده می‌شود.

### ****عملکرد کلی:****

* این مالتی‌پلکسر بین دو ورودی in0 و in1 انتخاب می‌کند.
* اگر سیگنال → sel = 0 خروجی برابر in0 خواهد بود.
* اگر سیگنال → sel = 1 خروجی برابر in1 خواهد بود.
* انتخاب به صورت **ترکیبی (Combinational)** انجام می‌شود و نیازی به کلاک ندارد.

### ****ورودی‌ها و خروجی‌ها:****

* :in0 داده ورودی اول.
* :in1 داده ورودی دوم.
* : sel سیگنال انتخاب – تعیین می‌کند کدام ورودی به خروجی وصل شود.
* : outخروجی انتخاب‌شده – یکی از دو ورودی به طور مستقیم در خروجی قرار می‌گیرد.

### ****نقش در Data Path:****

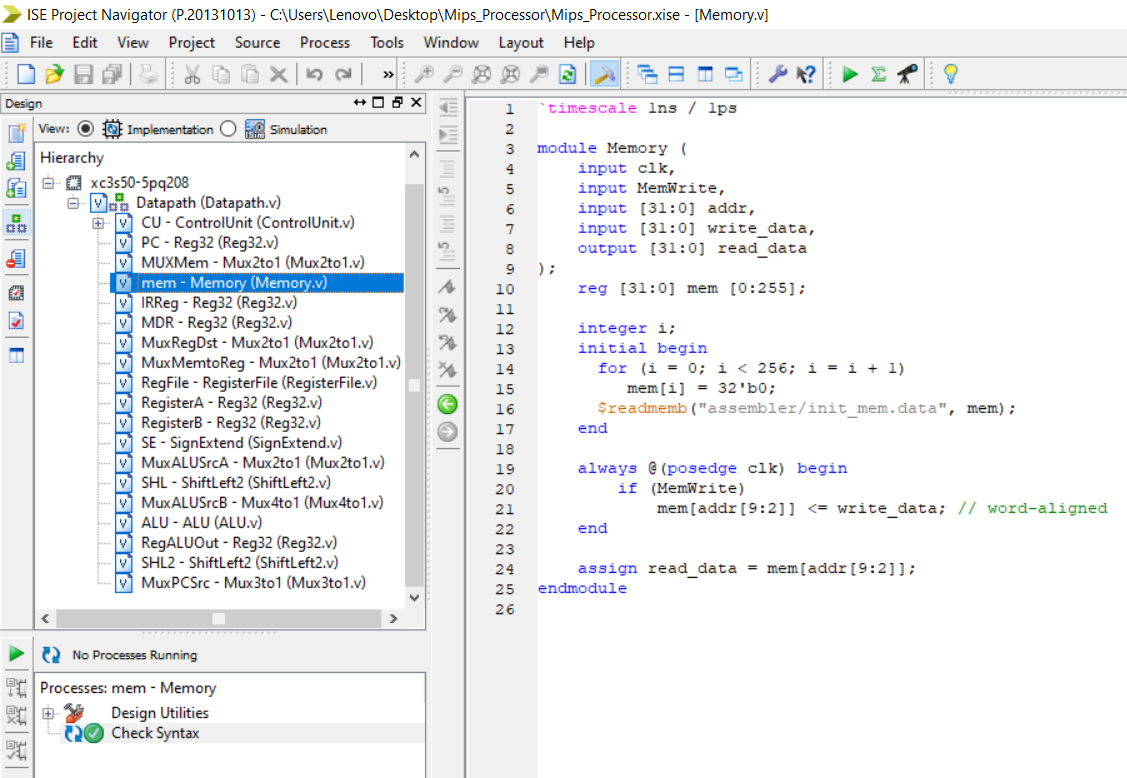
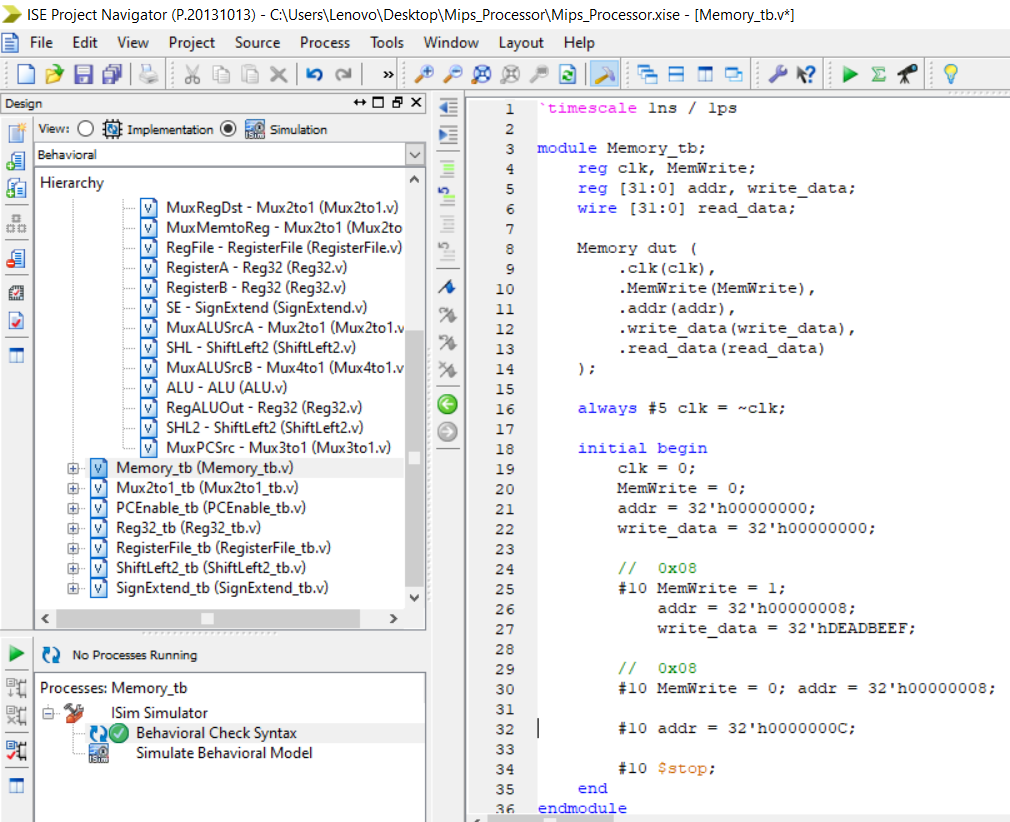
* این نوع MUX در چند نقطه از مسیر داده MIPS استفاده می‌شود:
  + انتخاب **PC+4** یا **Target Address** برای ورودی .PC
  + انتخاب بین **خروجی ALU** و **داده حافظه** برای نوشتن در فایل رجیستر . (WB stage)
  + انتخاب بین **Immediate** و **رجیستر B** برای ورودی. ALU
* چون این ماژول **پارامتریک** است، می‌تواند برای سیگنال‌های ۳۲ بیتی، ۵ بیتی یا هر پهنایی به‌کار رود.

### ****ارتباط با مراحل اجرای دستور در MIPS چند‌سیکل:****

* **:** **IF**انتخاب آدرس بعدی) PC معمولاً بین PC+4 و آدرس شاخه(.
* **: EX**انتخاب داده مناسب برای ورودی ALU.
* **:WB** انتخاب داده مناسب برای نوشتن در رجیستر مقصد.

### ****نام ماژول Memory :****

**نوع: حافظه داده/دستور ۳۲ بیتی با ۲۵۶ کلمه (word)**



این ماژول حافظه‌ی اصلی پردازنده‌ی MIPS چند‌سیکل است. این حافظه هم برای **دستورات (Instruction Memory)** و هم برای **داده‌ها (Data Memory)** استفاده می‌شود.

### ****عملکرد کلی:****

* حافظه شامل **۲۵۶ خانه‌ی ۳۲ بیتی** است (در مجموع ۱ کیلوبایت).
* در هنگام شبیه‌سازی، محتوای حافظه از یک فایل خارجی "assembler/init\_mem.data" بارگذاری می‌شود.
* عملیات **خواندن** داده به صورت ترکیبی (بدون کلاک) انجام می‌شود.
* عملیات **نوشتن** داده تنها در لبه‌ی بالارونده کلاک و در صورت فعال بودن MemWrite انجام می‌شود.
* آدرس‌دهی **word-aligned** است، یعنی از بیت‌های [9:2] آدرس برای انتخاب خانه حافظه استفاده می‌شود (چون هر خانه ۴ بایت است و دو بیت پایینی صفر هستند).

### ****ورودی‌ها و خروجی‌ها:****

* **ورودی‌ها:**
  + : clk سیگنال کلاک برای هماهنگی نوشتن.
  + :MemWrite اگر ۱ باشد، داده در حافظه نوشته می‌شود.
  + :addr آدرس ۳۲ بیتی (تنها بیت‌های ۹ تا ۲ استفاده می‌شوند).
  + : write\_data داده‌ای که قرار است نوشته شود.
* **خروجی:**
  + : read\_data داده‌ای که از حافظه خوانده می‌شود.

### ****نقش در Data Path:****

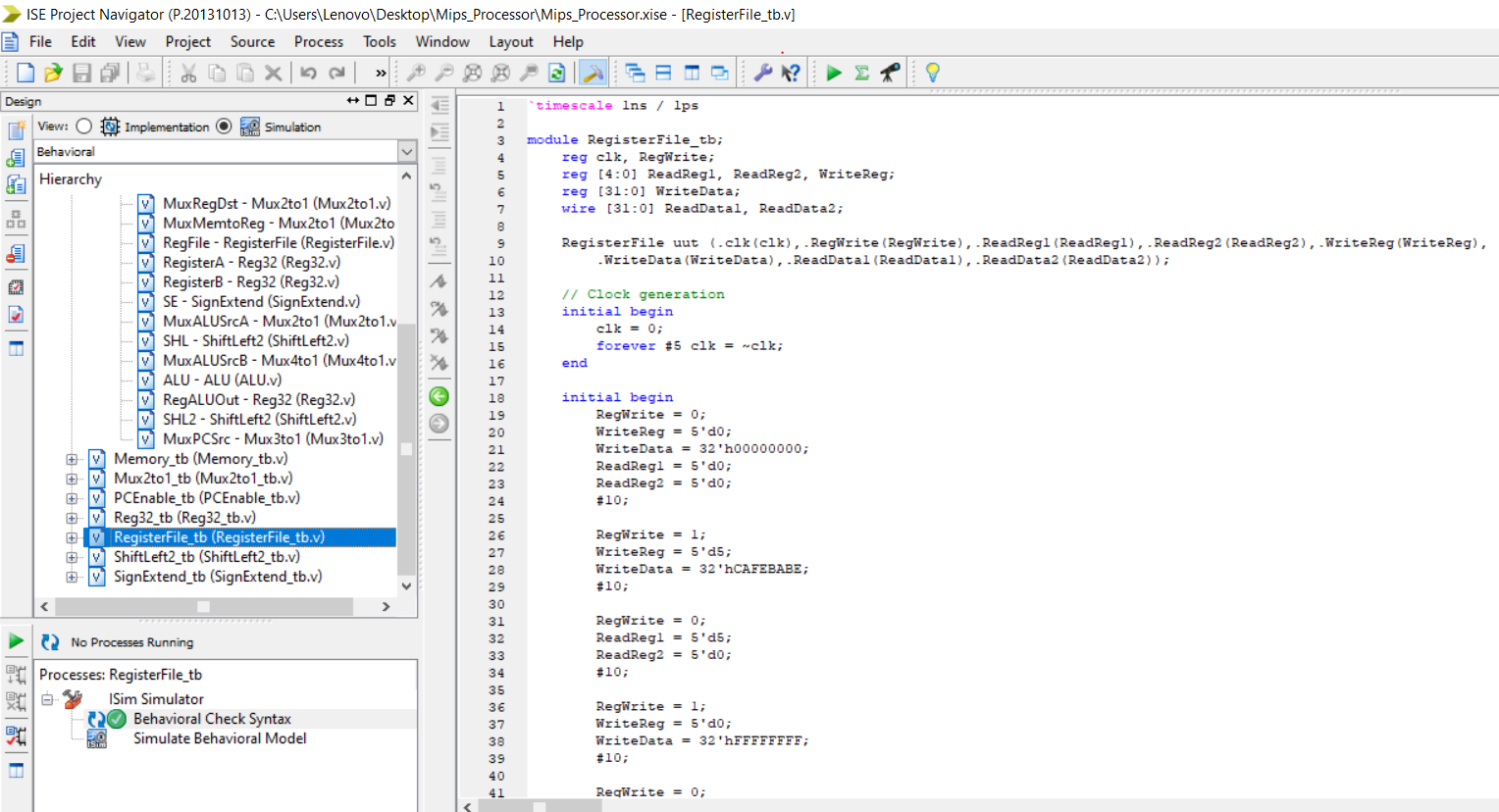
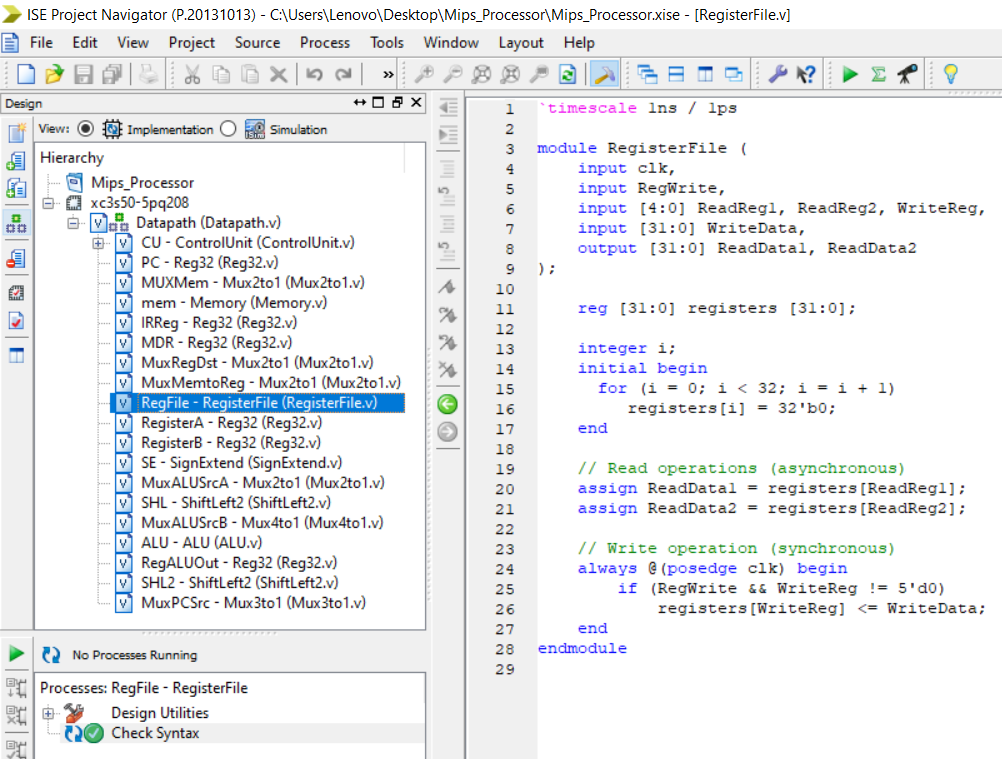
* در مرحله‌ی **:IF (Instruction Fetch)** پردازنده دستور را از همین حافظه می‌خواند.
* در مرحله‌ی **:** **MEM (Memory Access)** اگر دستور lw یا sw باشد، پردازنده داده را از همین حافظه می‌خواند یا در آن می‌نویسد.

### ****ویژگی‌های مهم:****

1. **مقداردهی اولیه حافظه**
   * با حلقه‌ی for تمام خانه‌های حافظه صفر می‌شوند.
   * سپس با $readmemb مقادیر برنامه‌ی اسمبلی از فایل بارگذاری می‌شوند.
2. **خواندن بدون کلاک و نوشتن با کلاک**
   * read\_data همیشه مقدار لحظه‌ای خانه‌ی حافظه انتخاب‌شده را بازتاب می‌دهد.
   * نوشتن تنها در لبه‌ی بالارونده کلاک و با MemWrite = 1 انجام می‌شود.
3. **آدرس‌دهی Word-Aligned**
   * برای دسترسی به حافظه ۳۲ بیتی، دو بیت پایین آدرس (addr[1:0]) نادیده گرفته می‌شوند.

### ****نام ماژول RegisterFile:****

**نوع: فایل رجیستر ۳۲×۳۲ (۳۲ رجیستر هرکدام ۳۲ بیتی)**



### این ماژول **Register File** پردازنده‌ی MIPS است و دقیقاً مطابق ساختار ۳۲ رجیستری MIPS طراحی شده است.

### ****عملکرد کلی:****

* شامل **۳۲ رجیستر عمومی R0) تا (R31** هرکدام ۳۲ بیتی است.
* قابلیت **خواندن همزمان از دو رجیستر** و **نوشتن در یک رجیستر** را دارد.
* رجیستر **R0 همیشه صفر است** و نمی‌توان در آن نوشت.
* **خواندن داده‌ها غیرهمزمان (asynchronous)** انجام می‌شود.
* **نوشتن داده‌ها همزمان با کلاک (synchronous)** و فقط وقتی RegWrite=1 باشد انجام می‌شود.

### ****ورودی‌ها و خروجی‌ها:****

* **ورودی‌ها:**
  + :clk سیگنال کلاک برای کنترل نوشتن
  + :RegWrite اگر ۱ باشد، داده در رجیستر مشخص‌شده نوشته می‌شود
  + : ReadReg1 آدرس رجیستری که باید روی پورت خواندن اول قرار گیرد
  + :ReadReg2 آدرس رجیستری که باید روی پورت خواندن دوم قرار گیرد
  + :WriteReg آدرس رجیستری که باید در آن نوشت
  + :WriteData داده‌ای که باید نوشته شود
* **خروجی‌ها:**
  + :ReadData1 مقدار رجیستر اول انتخاب‌شده
  + :ReadData2 مقدار رجیستر دوم انتخاب‌شده

### ****نقش در Data Path:****

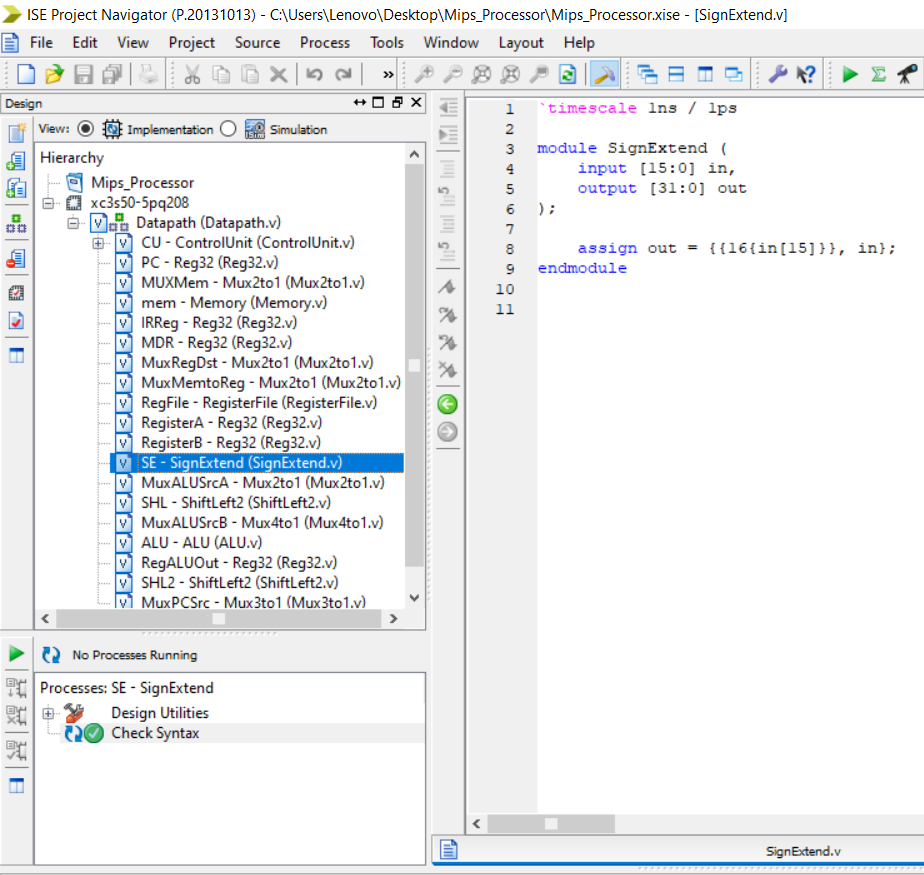
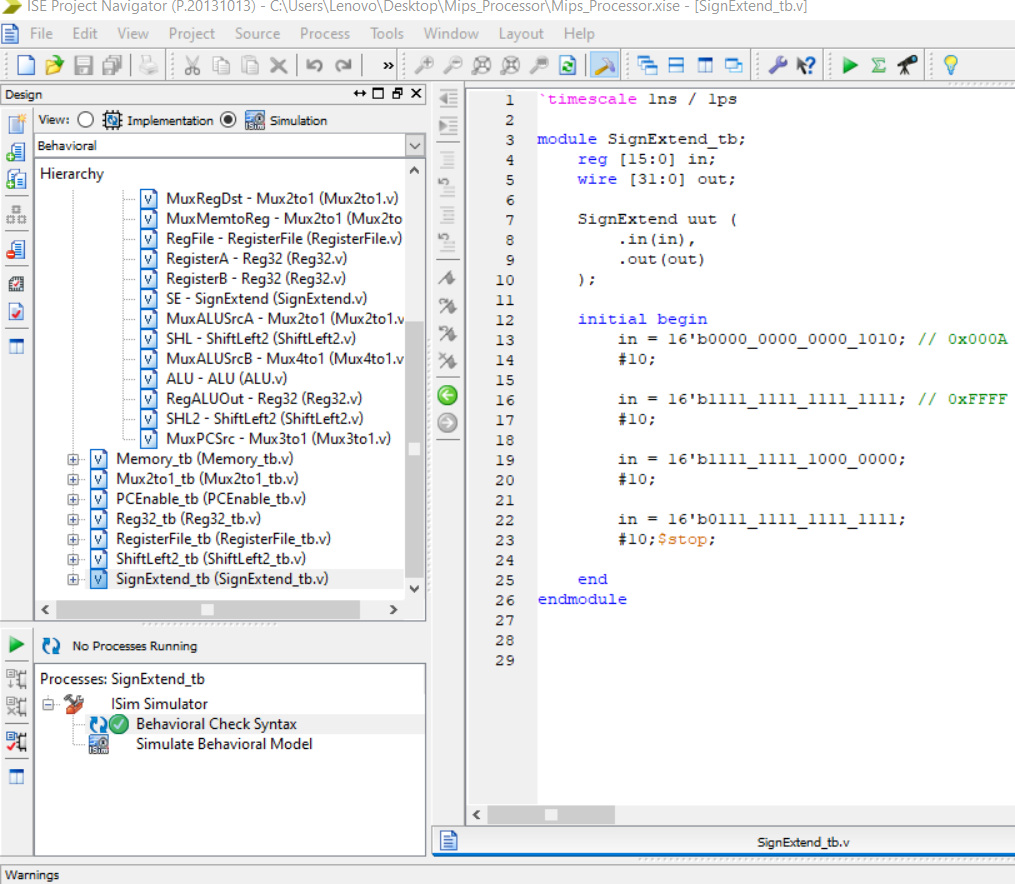
* در مرحله‌ی **:ID (Instruction Decode)** شماره‌ی رجیسترهای خوانده‌شده از فیلدهای rs و rt گرفته می‌شود و مقادیر آن‌ها به ترتیب به **ReadData1** و **ReadData2** ارسال می‌شود.
* در مرحله‌ی **:WB (Write Back)** اگر دستور نیاز به نوشتن نتیجه داشته باشد (مثل add, lw)، داده از طریق **WriteData** در رجیستر مقصد (WriteReg) ذخیره می‌شود.

### ****ویژگی‌های مهم:****

1. **مقداردهی اولیه:**
   * در ابتدای شبیه‌سازی، تمام ۳۲ رجیستر صفر می‌شوند.
2. **خواندن غیرهمزمان:**
   * هر تغییری در آدرس‌های ورودی ReadReg1 یا ReadReg2 فوراً روی خروجی‌ها بازتاب می‌شود (بدون نیاز به کلاک).
3. **نوشتن همزمان و جلوگیری از تغییر :R0**
   * فقط در لبه بالارونده کلاک و وقتی RegWrite=1 فعال باشد انجام می‌شود.
   * دستور شرطی WriteReg != 5'd0 مانع از نوشتن در رجیستر صفر می‌شود.

### ****نام ماژول SignExtend:****

**نوع: توسعه‌دهنده‌ی علامت از ۱۶ بیت به ۳۲ بیت**



این ماژول **SignExtend** برای توسعه‌ی علامت (Sign Extension) استفاده می‌شود

### ****عملکرد کلی:****

* وظیفه‌ی این ماژول این است که عدد ۱۶ بیتی **با علامت** را به عدد ۳۲ بیتی تبدیل کند و علامت آن را حفظ کند.
* بیت پرارزش (MSB) ورودی in[15] به‌عنوان بیت علامت در نظر گرفته می‌شود و در بیت‌های بالایی تکرار می‌شود.
* اگر →in[15] = 0 عدد مثبت بوده و 16 بیت بالایی با صفر پر می‌شود.
* اگر → in[15] = 1عدد منفی بوده و 16 بیت بالایی با یک پر می‌شود (حفظ علامت).

### ****ورودی‌ها و خروجی‌ها:****

* **ورودی:**
  + → in [15:0] عدد ۱۶ بیتی با علامت از دستوراتی مثل addi, lw, sw, beq
* **خروجی:**
  + → out [31:0] نسخه‌ی ۳۲ بیتی همان عدد با حفظ علامت

### ****نقش در Data Path:****

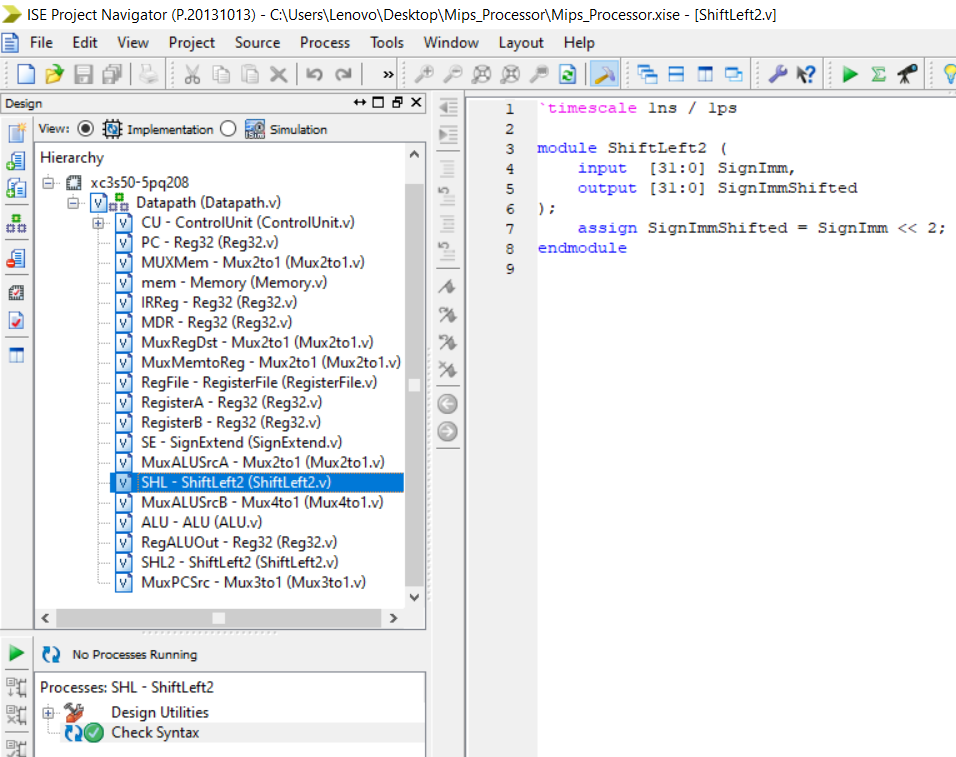
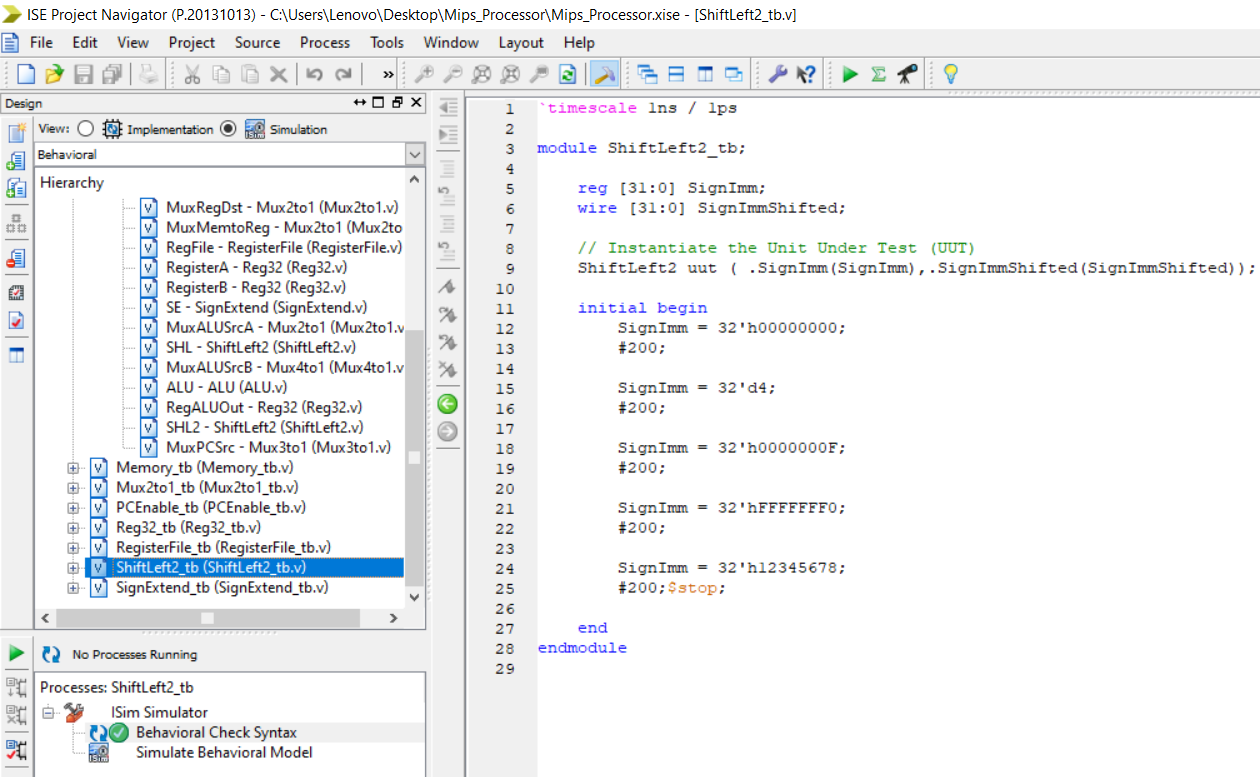
* در پردازنده‌ی MIPS، بیشتر دستورات Immediate مثل (**addi, lw, sw, beq**) دارای فیلد immediate 16 بیتی هستند.
* این ماژول مقدار ۱۶ بیتی را به ۳۲ بیتی تبدیل می‌کند تا بتواند مستقیماً وارد ALU یا سایر واحدها شود.

### ****ویژگی‌های مهم:****

1. **کاملاً ترکیبی (Combinational):**
   * هیچ کلاک یا لچ ندارد، به محض تغییر ورودی خروجی به‌روزرسانی می‌شود.
2. **بدون تأخیر اضافی:**
   * صرفاً اتصال سیمی است و از عملگر الحاق {} برای پر کردن بیت‌های بالا استفاده شده.

### ****نام ماژول ShiftLeft2:****

**نوع: شیفت‌دهنده‌ی ترکیبی (Combinational Shifter)**



این ماژول **ShiftLeft2** در مسیر داده‌ی پردازنده‌ی MIPS برای شیفت دادن مقادیر Immediate به کار می‌رود.

### ****عملکرد کلی:****

* این ماژول عدد ۳۲ بیتی ورودی را **دو بیت به چپ شیفت می‌دهد**) معادل ضرب در ۴ (.
* معمولاً برای محاسبه‌ی آدرس پرش‌ها یا شاخه‌ها (Branch) استفاده می‌شود، چون در MIPS آدرس‌ها Word- aligned هستند.
* انتقال تمام بیت‌های ورودی دو پله به چپ است.
* دو بیت کم‌ارزش صفر می‌شود.
* نتیجه در خروجی ظاهر می‌شود و هیچ تاخیر کلاکی ندارد.

**ورودی‌ها و خروجی‌ها:**

* **ورودی:**
  + → SignImm [31:0] معمولاً خروجی ماژول SignExtend است.
* **خروجی:**
  + → SignImmShifted [31:0] مقدار ورودی در ۴ ضرب شده است.

**نقش در Data Path:**

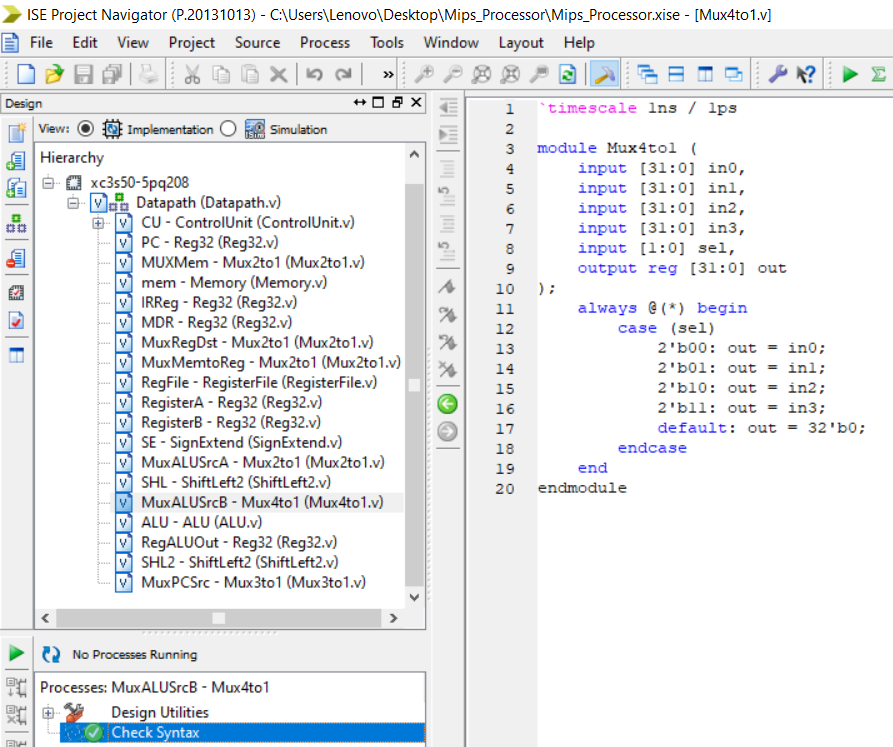
* در دستورات شاخه (**beq**, **bne**) باید آفست ۱۶ بیتی به ۳۲ بیت گسترش یافته و سپس ۴ برابر شود چون آدرس دستورها به صورت word (۴ بایت) هستند.
* همچنین در محاسبه آدرس پرش‌های jump و jal نیز برای هم‌ترازی به کار می‌رود.

**ویژگی‌های مهم:**

1. **کاملاً ترکیبی (Combinational):**
   * بدون لچ یا فلیپ‌فلاپ، خروجی فوراً تغییر می‌کند.
2. **ساده و بهینه:**
   * فقط سیم‌کشی داخلی است و نیازی به منطق اضافی ندارد.

### ****نام ماژول: Mux4to1****

**نوع: مالتی‌پلکسر ۴ به ۱**



### ****عملکرد کلی:****

* این ماژول **یکی از چهار ورودی ۳۲ بیتی را بر اساس سیگنال انتخاب sel به خروجی هدایت می‌کند.**
* در پردازنده‌ی MIPS Multi-Cycle برای انتخاب منبع ورودی‌های ALU یا داده‌های نوشتنی در رجیسترها به کار می‌رود.

**ورودی‌ها و خروجی‌ها:**

* **ورودی‌ها:**
  + → in0, in1, in2, in3 داده‌های ۳۲ بیتی
  + → sel[1:0] سیگنال انتخاب (کنترلی)
* **خروجی:**
  + → out[31:0] یکی از ورودی‌ها بر اساس sel

**نقش در Data Path:**

* در **واحد ALU**، این ماژول می‌تواند یکی از منابع مختلف را برای ورودی انتخاب کند.
* همچنین در **مرحله Write-back** رجیسترها، می‌تواند تعیین کند داده از حافظه، ALU یا PC+4 انتخاب شود.
* به‌صورت کلی هر جا که باید از بین چند مسیر مختلف فقط یکی انتخاب شود، این ماژول ضروری است.

**ویژگی‌های مهم:**

1. **کاملاً ترکیبی (بدون تاخیر کلاک)** → نتیجه انتخاب فوراً روی خروجی ظاهر می‌شود.
2. **قابل توسعه با پارامتر** → می‌توان نسخه پارامتری ساخت تا پهنای بیت تغییر کند.
3. **رفتار امن در حالت default** → جلوگیری از لاچ ناخواسته و خروجی ناشناخته.

### ****نام ماژول ALU (Arithmetic Logic Unit) :****

**نوع: واحد محاسباتی – منطقی) ترکیبی(Combinational–**





### ****عملکرد کلی:****

* این ماژول **عملیات اصلی پردازنده MIPS Multi-Cycle** مانند AND ، OR، جمع، تفریق و مقایسه Set Less) (Than را انجام می‌دهد.
* بر اساس سیگنال کنترلی ALUControl[2:0] مشخص می‌شود که ALU چه عملیاتی روی دو ورودی SrcA و SrcB انجام دهد.
* سیگنال **Zero** زمانی فعال می‌شود که نتیجه ALU برابر صفر شود( برای دستورات شرطی مانند BEQ)

### ****ورودی‌ها و خروجی‌ها:****

* **ورودی‌ها:**
  + → SrcA[31:0] عملوند اول ALU
  + → SrcB[31:0] عملوند دوم ALU
  + → ALUControl[2:0] سیگنال کنترل برای تعیین نوع عملیات
* **خروجی‌ها:**
  + → ALUResult[31:0] نتیجه عملیات محاسباتی/منطقی
  + → Zero یک بیت نشانگر صفر بودن نتیجه

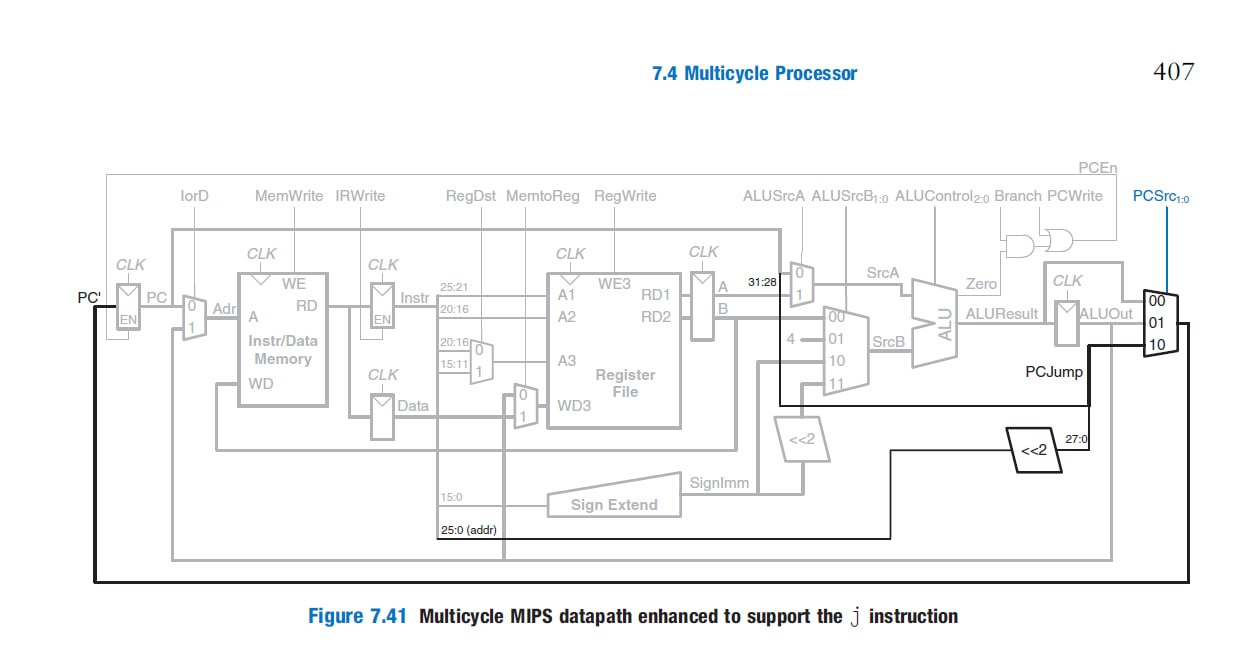
**نقش در Data Path:**

* در **مرحله Execute** دستورالعمل‌ها، ALU محاسبات آدرس حافظه، نتیجه عملیات منطقی یا جمع/تفریق را انجام می‌دهد.
* سیگنال **Zero** برای اجرای پرش‌های شرطی (BEQ, BNE) در کنترل واحد استفاده می‌شود.

**ویژگی‌های مهم:**

1. **کاملاً ترکیبی و بدون تاخیر کلاک** → خروجی سریع به‌روز می‌شود.
2. **پشتیبانی از دستورات اصلی R-Type و I-Type** در پردازنده. MIPS
3. **قابل توسعه برای عملیات بیشتر** مثل XOR ، NOR و شیفت‌ها.
4. **پرچم Zero داخلی** → ساده‌سازی کنترل واحد.

**ماژول — Mux3to1 انتخاب ورودی برای PC**



به datapathای که داشتیم یک Mux 3to1 اضافه میکنیم تا دیتاپس بتواند دستور jump را هم پشتیبانی کند.

**نقش ماژول در پردازنده:**

* این مالتی‌پلکسر تعیین می‌کند **مقدار بعدی PC** از کجا بیاید:
  + : **in0 (PC+4) → 00**حالت عادی اجرای متوالی دستورها
  + : **in1 (Branch)** **→ 01**وقتی شرط پرش برقرار است.
  + **in2 (Jump)** **→10**:وقتی دستور J یا JAL اجرا می‌شود

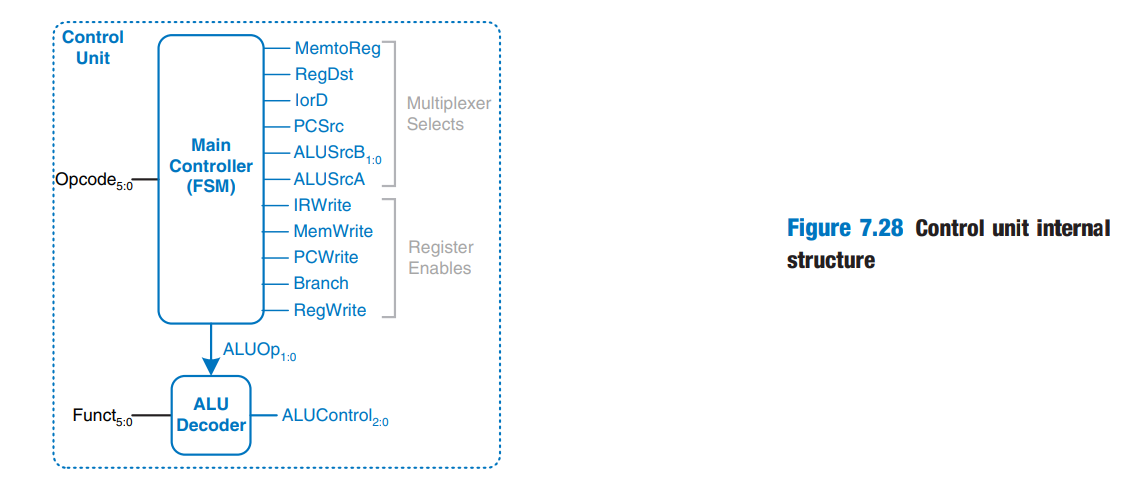
**جزئیات پیاده‌سازی:**

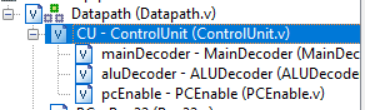
* **ورودی‌ها:** سه عدد ۳۲ بیتی که هر کدام منبعی برای آدرس بعدی PC هستند.
* **)sel سیگنال انتخاب:** **(**۲ بیت از کنترل واحد که مشخص می‌کند کدام ورودی انتخاب شود.
* **خروجی:** یک عدد ۳۲ بیتی که مستقیماً به رجیستر PC وصل می‌شود.

**چرا در معماری چندسیکل MIPS نیاز داریم؟**

* چون **PC همیشه از یک مسیر نمی‌آید**:
  + مسیر PC+4 برای ادامه اجرای معمولی
  + مسیر) Branch برای دستورهای شرطی مثل BEQ )
  + مسیر Jump برای پرش مستقیم مثل (J, JAL)
* اگر این مالتی‌پلکسر نباشد، PC فقط یک ورودی دارد و امکان پرش وجود نخواهد داشت.

**Control Unit:**





**Control Unit (CU)** یکی از اصلی‌ترین بخش‌های پردازنده است که وظیفه تولید و مدیریت سیگنال‌های کنترلی را بر عهده دارد. این سیگنال‌ها مسیر جریان داده (data path) و رفتار اجزای مختلف پردازنده مثل ALU، Register File، Memory و PC را مشخص می‌کنند. در معماری **MIPS multicycle processor**، اجرای هر دستور در چندین سیکل کلاک انجام می‌شود ) برخلاف single-cycle که همه چیز در یک سیکل اتفاق می‌افتد. ( به همین دلیل، واحد کنترل باید بداند که در هر **مرحله** چه عملی باید صورت بگیرد.

**Control Unit مغز پردازنده است**. این واحد به دستور نگاه می‌کند، تشخیص می‌دهد باید چه کاری انجام شود، و سپس با فعال‌سازی یا غیرفعال‌سازی سیگنال‌ها، اجزای مختلف پردازنده را هماهنگ می‌کند.

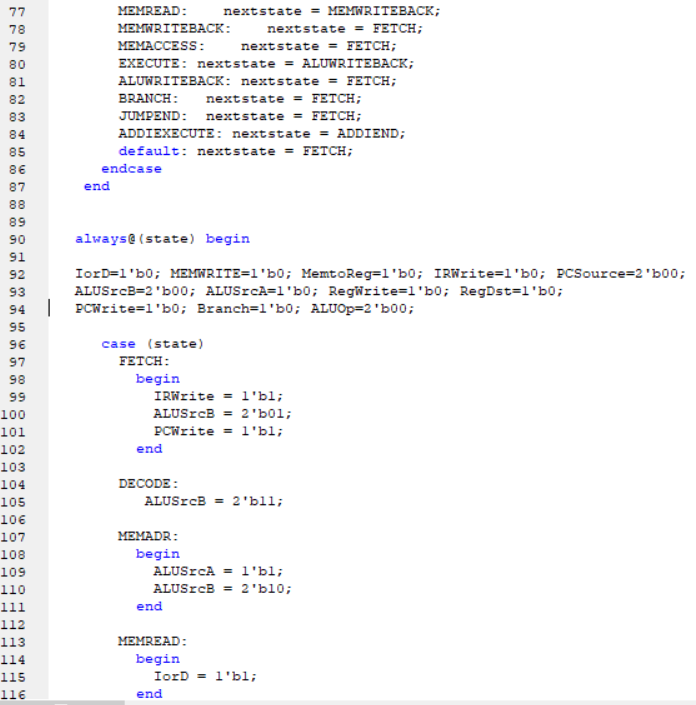
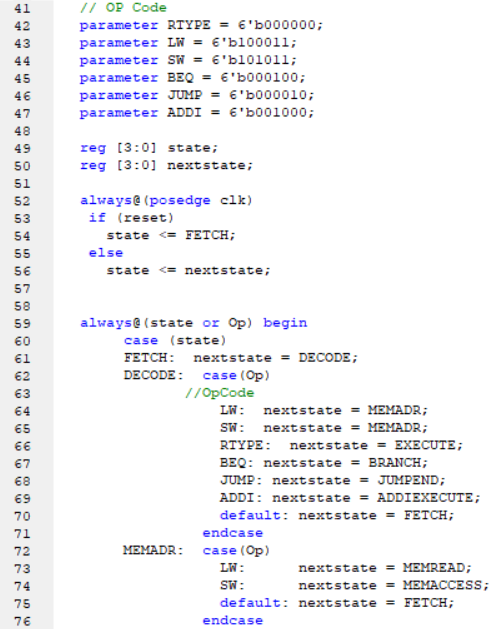
Control Unit از سه جزء اصلی تشکیل شده است:

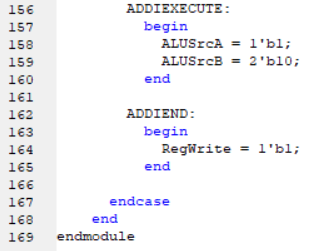
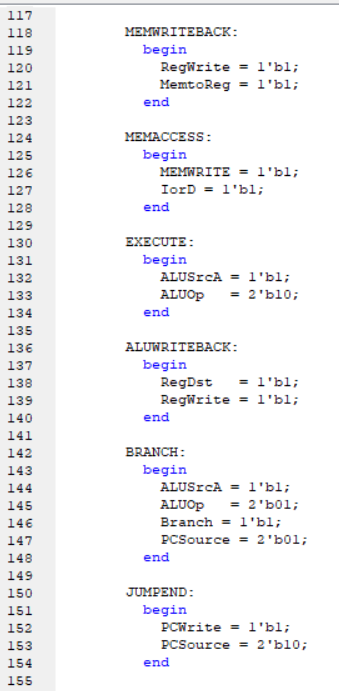
1. :**MainDecoder (FSM)** تعیین توالی وضعیت‌ها و تولید سیگنال‌های کنترلی سطح‌بالا بر اساس Op .
2. :**ALUDecoder** دیکود ALUOp و) برای (R-typeفیلد Funct به سیگنال نهایی ALUCtrl .
3. :**PCEnable** گیت کردن نوشتن روی PC با توجه به PCWrite و شرط Branch (Branch & Zero).

ماژول **ControlUnit** این سه جزء را یکپارچه می‌کند و سیگنال‌های نهایی دیتاپث را می‌سازد.

## **1) MainDecoder (FSM)**







* **Inputs**:  
  clk, reset, Op[5:0]
* **Outputs (Datapath controls)**:  
  IorD, RegDst, PCSource[1:0], MemtoReg, ALUSrcA, ALUSrcB[1:0],  
  IRWrite, MEMWRITE, RegWrite, PCWrite, Branch, ALUOp[1:0]

### State Encoding:

### FETCH=0000, DECODE=0001, MEMADR=0010, MEMREAD=0011, MEMWRITEBACK=0100,

### MEMACCESS=0101, EXECUTE=0110, ALUWRITEBACK=0111, BRANCH=1000,

### JUMPEND=1001, ADDIEXECUTE=1010, ADDIEND=1011

### Opcode Map:

### RTYPE=000000, LW=100011, SW=101011, BEQ=000100, JUMP=000010, ADDI=001000

### Transition Logic (خلاصه):

* FETCH → DECODE (همیشه)
* DECODE → MEMADR برای LW/SW
* DECODE → EXECUTE برای RTYPE
* DECODE → BRANCH برای BEQ
* DECODE → JUMPEND برای J
* DECODE → ADDIEXECUTE برای ADDI
* سایر گذارها مطابق کد، همگی به FETCH باز می‌گردند.

### Output Logic per State (Micro-ops):

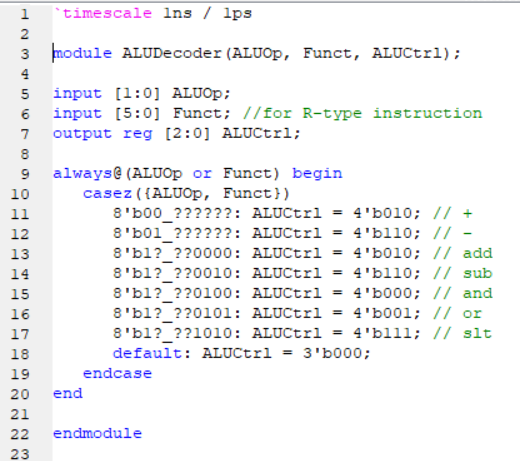
نکته: در ابتدای بلوک خروجی‌ها همه‌ی سیگنال‌ها صفر می‌شوند تا latch شکل نگیرد.

* **FETCH :**
  + ) IRWrite=1 لود دستور به (IR
  + )ALUSrcB=01 ثابت ALU = PC + 4 ⇒ ( 4
  + PCWrite=1, PCSource=00 ⇒ PC ← PC+4
  + )IorD=0 آدرس حافظه از( PC
* **DECODE:**
  + ALUSrcB=11 ⇒ ALU = PC + (SignImm << 2) (محاسبه Branch target در ALUOut )
* **MEMADR** برای (LW/SW)
  + ALUSrcA=1 (A), ALUSrcB=10 (SignImm) ⇒ EA = A + SignImm
* **MEMREAD** )مرحله خواندن LW)
  + ⇒ IorD=1 آدرس حافظه از ALUOut
* ) **MEMWRITEBACK** پایان (LW
  + RegWrite=1, MemtoReg=1 ⇒ rt ← MDR
* )  **MEMACCESS** برای (SW
  + IorD=1, MEMWRITE=1 ⇒ Mem[ALUOut] ← RegB
* **EXECUTE** (R-type)
  + ) ALUSrcA=1, ALUOp=10 انتخاب عمل از) ALUSrcB=00( Funct ضمنی(
* ) **ALUWRITEBACK** پایان R-type)
  + RegDst=1, RegWrite=1, MemtoReg=0 ⇒ rd ← ALUResult
* **BRANCH** (BEQ)
  + ALUSrcA=1, ALUOp=01 ⇒ ALU = A - B
  + ⇒ Branch=1, PCSource=01 اگر PC ، Zero=1به Branch target می‌پرد.
* **JUMPEND** (J)
  + PCWrite=1, PCSource=10 ⇒ PC ← JumpAddress
* **ADDIEXECUTE / ADDIEND**
  + ADDIEXECUTE: ALUSrcA=1, ALUSrcB=10, ALUOp=00 ⇒ A + SignImm
  + ADDIEND: RegWrite=1, RegDst=0, MemtoReg=0 ⇒ rt ← ALUResult

### نکات طراحی:

* **Reset** در کد شما synchronous است (داخل ((posedge clk) @، و state روی FETCH ست می‌شود.
* مقداردهی پیش‌فرض خروجی‌ها در ابتدای بلوک always@(state) از ایجاد latch جلوگیری می‌کند.
* برای opcodeهای نامعتبر، ماشین حالت به FETCH برمی‌گردد (safe default).

**2) ALUDecoder:**



* **Inputs**: ALUOp[1:0], Funct[5:0]
* **Output**: ALUCtrl[2:0] (ALU: 000=AND, 001=OR, 010=ADD, 110=SUB, 111=SLT نگاشت به )

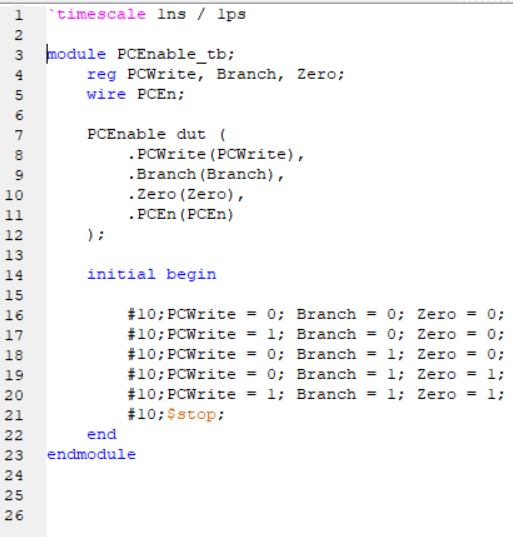
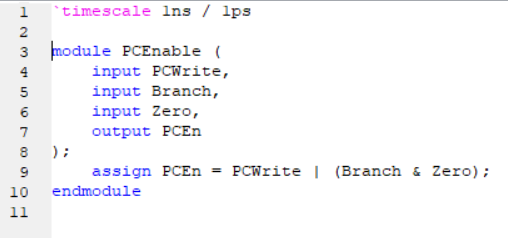
### Functionality:

* اگر ) ALUOp=00 → **ADD** برای (LW/SW/ADDI
* اگر ) ALUOp=01 → **SUB** برای (BEQ
* اگر → ALUOp=1x نوع عملیات از Funct (R-type):
  + ADD (010) → 100000
  + SUB (110) → 100010
  + AND (000) → 100100
  + OR (001) → 100101
  + SLT (111) → 101010

### هم‌خوانی با: ALU

کد ALU دقیقاً همین کدگذاری را انتظار دارد (AND=000, OR=001, ADD=010, SUB=110, SLT=111)، بنابراین خروجی این دیکودر با ALU هم‌راستاست.

## **3) PCEnable:**



 **Inputs**: PCWrite, Branch, Zero

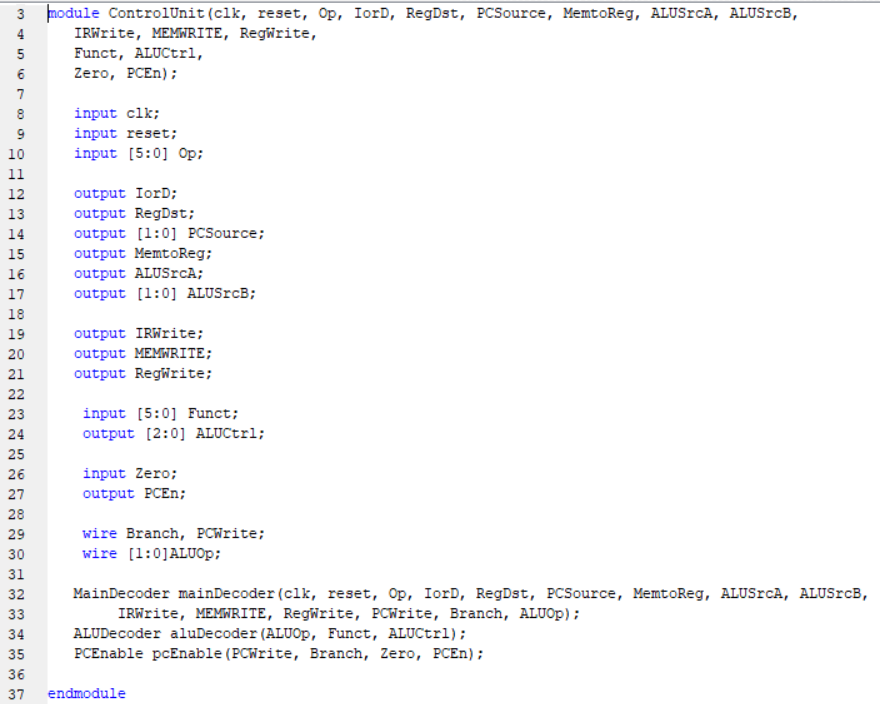
 **Output**: PCEn

 در **FETCH** و ) **JUMP** ,PCWrite=1 ⇒ PCEn=1 بدون شرط **(.**

 در **BRANCH**، فقط اگر نتیجه ALU صفر شود (Zero=1) و Branch=1، آنگاه PC به آدرس Branch به‌روزرسانی می‌شود.

 در سایر حالت‌ها PCEn=0، بنابراین PC لچ نمی‌شود.

**4) ControlUnit (Integration):**



### وظایف اصلی :Control Unit

1. **Decoding دستور** : بعد از اینکه یک دستور از حافظه واکشی (fetch) شد، واحد کنترل کد عملیاتی (opcode) را بررسی می‌کند و نوع دستور R-type)، I-type، (J-type را تشخیص می‌دهد.
2. **تولید سیگنال‌های کنترلی** : بر اساس نوع دستور و فاز اجرای آن (fetch, decode, execute, memory, write-back)، سیگنال‌هایی مثل:
   * ) RegWrite نوشتن در رجیسترها(
   * ) ALUSrc انتخاب ورودی (ALU
   * MemRead و MemWrite )خواندن/نوشتن از حافظه(
   * ) PCSrc تعیین مقدار جدید Program Counter برای (jump/branch
   * ) MemtoReg انتخاب داده برگشتی به رجیستر از ALU یا حافظه(
   * و سایر سیگنال‌ها را تولید می‌کند.
3. **مدیریت مراحل اجرای دستور**: CU به صورت یک **Finite State Machine (FSM)** عمل می‌کند. یعنی با توجه به حالت فعلی (state) و opcode، به حالت بعدی می‌رود و سیگنال‌های مناسب را فعال می‌کند.

### ساختار کلی:

* **Input ها**: شامل Opcode، بیت‌های خاصی از دستور مثل funct ) برای (R-typeو گاهی شرایط ) ALU مثل Zero (flag
* **Output ها**: سیگنال‌های کنترلی که به اجزای datapath داده می‌شوند.
* : **FSM**برای مدیریت سیکل‌های اجرای دستور.
* **Inputs**: clk, reset, Op[5:0], Funct[5:0], Zero
* **Outputs**:  
  Datapath controls: IorD, RegDst, PCSource[1:0], MemtoReg, ALUSrcA, ALUSrcB[1:0], IRWrite, MEMWRITE, RegWrite  
  ALU control: ALUCtrl[2:0]  
  PC gating: PCEn

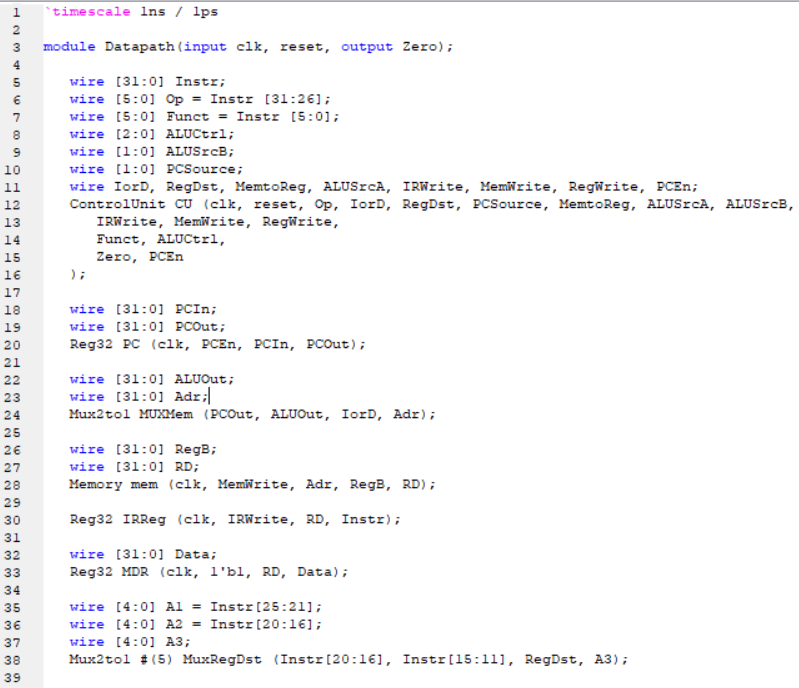
### اتصال داخلی:

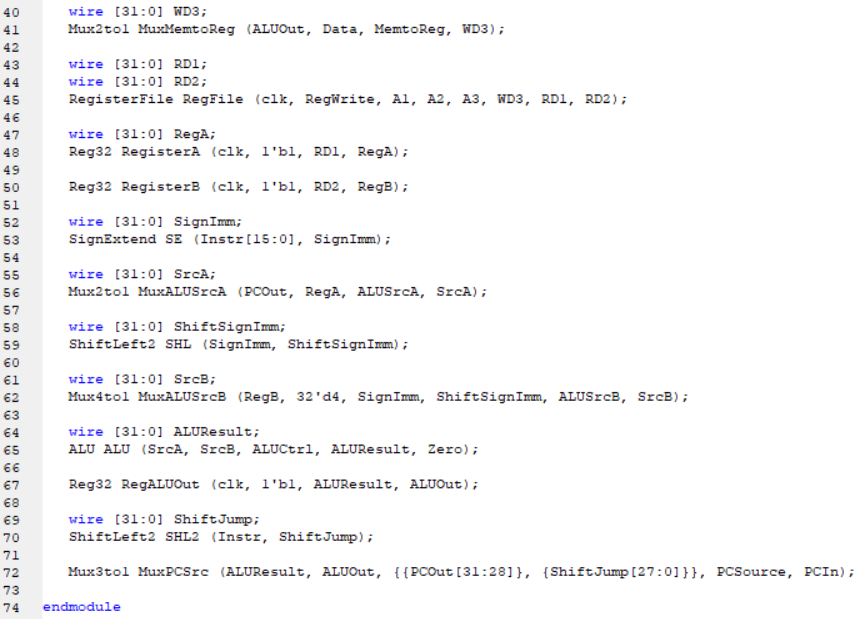
* MainDecoder سیگنال‌های سطح‌بالا ALUOp + را می‌دهد.
* ALUDecoder از ALUOp و Funct، ALUCtrl نهایی را تولید می‌کند.
* PCEnable با PCWrite, Branch, Zero ، سیگنال PCEn را برای رجیستر PC می‌سازد.

### توالی سیکل‌ها (نمونه):

* **LW**: FETCH → DECODE → MEMADR → MEMREAD → MEMWRITEBACK → FETCH
* **SW**: FETCH → DECODE → MEMADR → MEMACCESS → FETCH
* **R-type**: FETCH → DECODE → EXECUTE → ALUWRITEBACK → FETCH
* **BEQ**: FETCH → DECODE → BRANCH → FETCH
* **J**: FETCH → DECODE → JUMPEND → FETCH
* **ADDI**: FETCH → DECODE → ADDIEXECUTE → ADDIEND → FETCH

## **Datapath Module:**





ماژول **Datapath** مسئول پیاده‌سازی مسیر داده (Data Path) پردازنده MIPS در حالت **Multicycle** است. در این طراحی، هر دستورالعمل در چندین سیکل اجرا می‌شود و سیگنال‌های کنترلی توسط **Control Unit** تولید می‌شوند تا اجزای datapath به صورت هماهنگ عمل کنند.

این ماژول شامل اجزای اصلی مانند **Program Counter (PC)**، **Register File**، **Memory**، **ALU**، **Muxها** و رجیسترهای میانی (Pipeline Registers) است که داده‌ها را بین مراحل مختلف اجرای دستور نگه می‌دارند.

### ورودی‌ها و خروجی‌ها:

* **ورودی‌ها**:
  + :clk کلاک سیستم
  + :reset سیگنال ریست
* **خروجی‌ها**:
  + : Zero خروجی Zero از ALU برای پشتیبانی از دستورات شرطی (مثل beq )

**بخش‌های اصلی ماژول:**

#### **Control Unit**

ControlUnit CU (...);

کنترل یونیت سیگنال‌های کنترلی مثل IorD, RegDst, PCSource, ALUSrcA, ALUSrcB, RegWrite, MemWrite MemtoReg, و همچنین ALUCtrl را تولید می‌کند.

این سیگنال‌ها رفتار datapath را در هر سیکل تعیین می‌کنند.

#### **Program Counter (PC)**

Reg32 PC (clk, PCEn, PCIn, PCOut);

رجیستری 32 بیتی که آدرس دستورالعمل جاری را نگه می‌دارد.

: **PCOut** آدرس جاری دستور را نگه می‌دارد.

: **PCIn** مقدار بعدی PC است که توسط MuxPCSrc انتخاب می‌شود.

: **PCEn** کنترل‌کننده فعال‌سازی PC است (با توجه به PCWrite یا Branch)

#### **Memory.3**

Mux2to1 MUXMem (PCOut, ALUOut, IorD, Adr);

Memory mem (clk, MemWrite, Adr, RegB, RD);

شامل هر دو بخش Instruction Memory و Data Memory است.

* با استفاده از مالتی‌پلکسر MUXMem، آدرس حافظه یا از PCOut (برای Fetch دستور) و یا از ) ALUOutبرای دستورات ( Load/Store انتخاب می‌شود.
* ماژول Memory داده را از حافظه می‌خواند (RD) یا در آن می‌نویسد.

#### **Instruction Register (IR) . 4 و Memory Data Register (MDR)**

Reg32 IRReg (clk, IRWrite, RD, Instr);

Reg32 MDR (clk, 1'b1, RD, Data);

* **Instr** :دستور واکشی‌شده از حافظه را ذخیره می‌کند.
* :**MDR** داده خوانده‌شده از حافظه را نگه می‌دارد تا در مراحل بعدی استفاده شود )مثلاً در دستور. (LW

#### **Register File.5**

RegisterFile RegFile (...);

* رجیستر فایل سه بخش دارد:
  + دو خروجی خواندن RD1) و (RD2
  + یک ورودی نوشتن (کنترل‌شده با RegWrite)
* انتخاب رجیستر مقصد (A3) توسط RegDst انجام می‌شود.
* داده‌ای که در رجیستر نوشته می‌شود از طریق MemtoReg بین خروجی ALU و داده حافظه انتخاب می‌شود.

#### **.6 رجیسترهای میانی Register A) و( Register B**

Reg32 RegisterA (clk, 1'b1, RD1, RegA);

Reg32 RegisterB (clk, 1'b1, RD2, RegB);

* داده خوانده‌شده از رجیستر فایل در این رجیسترها ذخیره می‌شود تا در مراحل بعدی مثل ورودی (ALU) استفاده گردد.

#### **Sign Extend .7 و Shifter**

SignExtend SE (Instr[15:0], SignImm);

ShiftLeft2 SHL (SignImm, ShiftSignImm);

* **SignExtend** مقدار immediate را از 16 بیت به 32 بیت گسترش می‌دهد.
* **ShiftLeft2** مقدار) offset برای branch یا (jump را دو بیت به چپ شیفت می‌دهد.

#### **ALU. 8 و انتخاب ورودی‌ها**

Mux2to1 MuxALUSrcA (PCOut, RegA, ALUSrcA, SrcA);

Mux4to1 MuxALUSrcB (RegB, 32'd4, SignImm, ShiftSignImm, ALUSrcB, SrcB);

ALU ALU (SrcA, SrcB, ALUCtrl, ALUResult, Zero);

* ورودی اول ALU (SrcA) از بین PCOut یا RegA انتخاب می‌شود.
* ورودی دوم ALU (SrcB) می‌تواند یکی از چهار گزینه باشد:
  + مقدار RegB
  + عدد ثابت 4 ) برای افزایش( PC
  + immediate گسترش‌یافته
  + immediate شیفت‌داده‌شده) برای( branch
* ALU عملیات موردنظر) جمع، تفریق، and، or، slt و غیره (را انجام می‌دهد.

#### **ALUOut Register.9**

Reg32 RegALUOut (clk, 1'b1, ALUResult, ALUOut);

* خروجی ALU در این رجیستر ذخیره می‌شود تا در سیکل‌های بعدی قابل استفاده باشد مثلاً در دستو. (Load/Store)

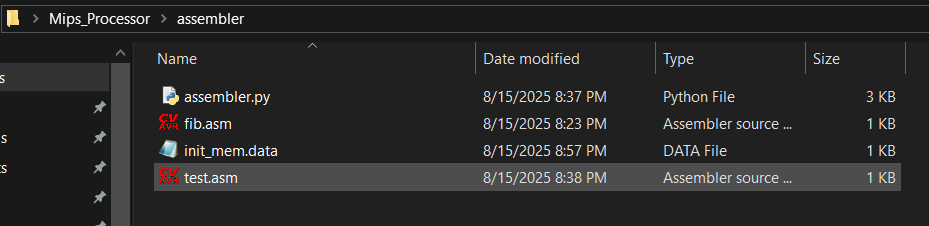
#### **Jump Address. 10و انتخاب PC**

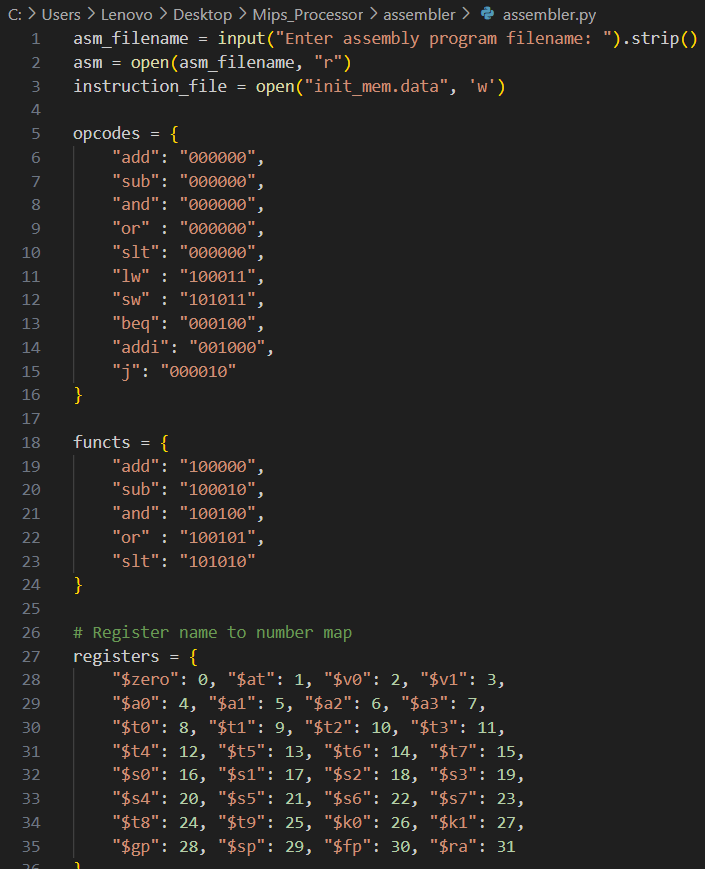
ShiftLeft2 SHL2 (Instr, ShiftJump);

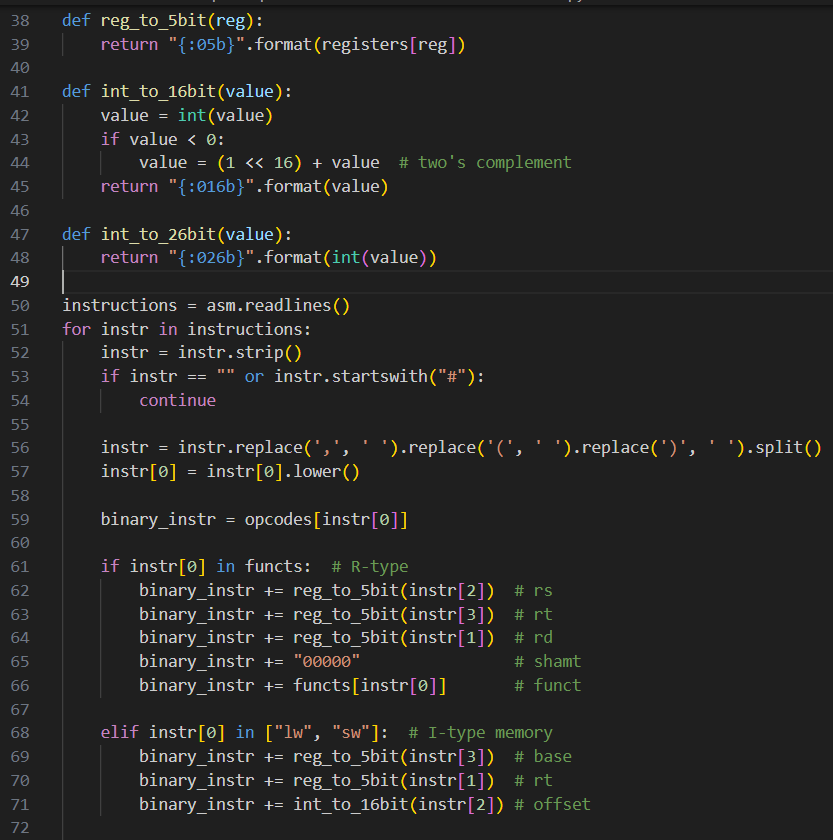
Mux3to1 MuxPCSrc (ALUResult, ALUOut, {{PCOut[31:28]}, {ShiftJump[27:0]}}, PCSource, PCIn);

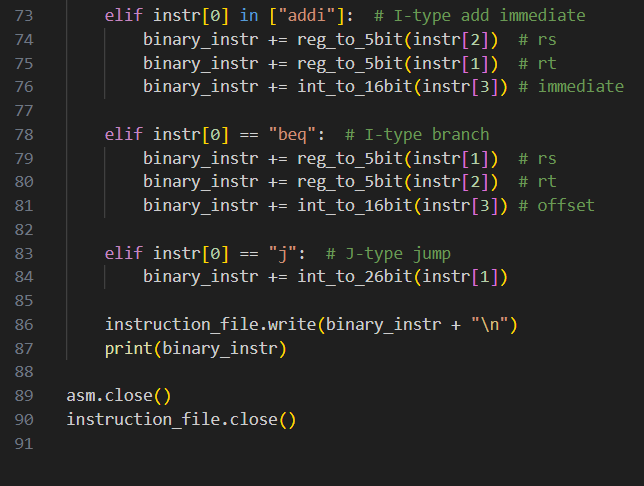
* برای پرش (Jump)، بخش 26 بیتی از دستور شیفت داده می‌شود و با بالاترین 4 بیت PC ترکیب می‌گردد.
* مالتی‌پلکسر MuxPCSrc یکی از سه گزینه را به عنوان مقدار بعدی PC انتخاب می‌کند:
  + ) ALUResult برای (PC+4
  + ALUOut برای( branch)
  + آدرس Jump
* ماژول **Datapath** در معماری MIPS چندسیکل، تمامی اجزای اصلی پردازنده را شامل می‌شود و با استفاده از سیگنال‌های کنترلی تولیدشده توسط **Control Unit**، وظیفه‌ی اجرای درست دستورالعمل‌ها را بر عهده دارد. ساختار ماژول به گونه‌ای است که داده‌ها به‌وسیله رجیسترهای میانی و مالتی‌پلکسرها بین اجزا حرکت کرده و مراحل Fetch، Decode، Execute، Memory Access و Write Back به صورت هماهنگ انجام می‌شوند.

# **Python Assembler Script تولید فایل( init\_mem.data)**









این اسکریپت در واقع یک **Assembler ساده** برای پردازنده‌ی MIPS است؛ یعنی کدی که فایل اسمبلی مثل (program.asm) را می‌گیرد و معادل باینری هر دستور را به صورت خط به خط داخل یک فایل حافظه (init\_mem.data) می‌نویسد. بعد همین فایل توسط ماژول **Memory** در Verilog ($readmemb) بارگذاری می‌شود.

## **هدف:**

* دریافت کد اسمبلی نوشته‌شده برای پردازنده‌ی MIPS
* ترجمه هر دستور به کد ماشین ۳۲ بیتی مطابق فرمت **R-type، I-type، J-type**
* ذخیره باینری‌ها در فایلی به نام init\_mem.data برای بارگذاری در حافظه دستور (Instruction Memory)
* این فایل بعداً در شبیه‌سازی (Memory.v) خوانده می‌شود تا پردازنده برنامه را اجرا کند.

## **ساختار کلی برنامه:**

### .1 خواندن ورودی و آماده‌سازی فایل خروجی

* از کاربر نام فایل اسمبلی (مثلاً (program.asm) گرفته می‌شود.
* فایل اسمبلی باز می‌شود و خط به خط خوانده خواهد شد.
* فایل خروجی init\_mem.data برای نوشتن باینری دستورها باز می‌شود.

### .2 تعریف جداول Opcode و Funct

* چون دستورات R-type همه Opcode = 0 دارند، ولی funct فرق می‌کند، این دو جدول جدا تعریف شده‌اند.
* دستورات I-type و J-type مثل (lw, sw, addi, beq, j) فقط از جدول opcodes استفاده می‌کنند.

### .3 تعریف جدول Registerها

* اسم رجیستر به شماره متناظر در معماری MIPS نگاشت می‌شود.
* مثال $t0 → 8: ، $s1 → 17

### 4.توابع کمکی

:**reg\_to\_5bit** شماره رجیستر را به رشته ۵ بیتی تبدیل می‌کند.

:**int\_to\_16bit** عدد Immediate را به ۱۶ بیت (signed, two’s complement) تبدیل می‌کند.

**:int\_to\_26bit** آدرس برای دستور j را به ۲۶ بیت تبدیل می‌کند.

### . 5پردازش خط به خط دستورها

 دستورها یکی‌یکی خوانده می‌شوند.

 خطوط خالی یا Comment با (#) رد می‌شوند.

 سپس با (',', ' ') replace. و ()split. کلمات دستور جدا می‌شوند.

### add $t2, $t0, $t1

### → ["add", "$t2", "$t0", "$t1"]

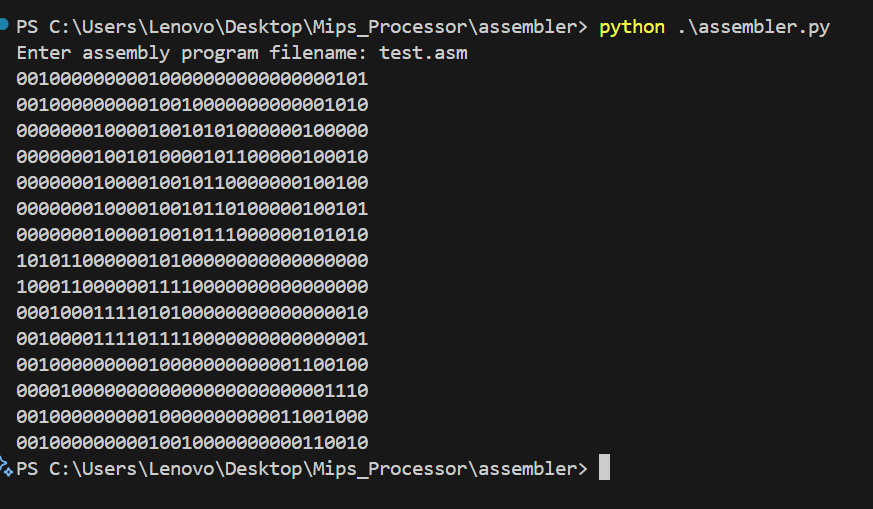
### .6 تولید کد باینری هر نوع دستور

### .7 نوشتن خروجی

 هر دستور ۳۲ بیتی به صورت یک خط در init\_mem.data نوشته می‌شود.

 همزمان در Console هم چاپ می‌شود ) برای (Debug

**فایل خروجی نمونه (init\_mem.data)**



این فایل نشان‌دهنده‌ی همان دستورات اسمبلی به صورت باینری است:

| Address | Instruction | Binary |

| ------- | ---------------------- | --------------------------------------- |

| 0 | addi $t0, $zero, 5 | 001000 00000 01000 0000000000000101 |

| 1 | addi $t1, $zero, 10 | 001000 00000 01001 0000000000001010 |

| 2 | add $t2, $t0, $t1 | 000000 01000 01001 01010 00000 100000 |

| 3 | sub $t3, $t1, $t0 | 000000 01001 01000 01011 00000 100010 |

| 4 | and $t4, $t0, $t1 | 000000 01000 01001 01100 00000 100100 |

| 5 | or $t5, $t0, $t1 | 000000 01000 01001 01101 00000 100101 |

| 6 | slt $t6, $t0, $t1 | 000000 01000 01001 01110 00000 101010 |

| 7 | sw $t2, 0($zero) | 101011 00000 01010 0000000000000000 |

| 8 | lw $t7, 0($zero) | 100011 00000 01111 0000000000000000 |

| 9 | beq $t7, $t2, 2 | 000100 01111 01010 0000000000000010 |

| 10 | addi $t7, $t7, 1 | 001000 01111 01111 0000000000000001 |

| 11 | addi $t0, $zero, 100 | 001000 00000 01000 0000000001100100 |

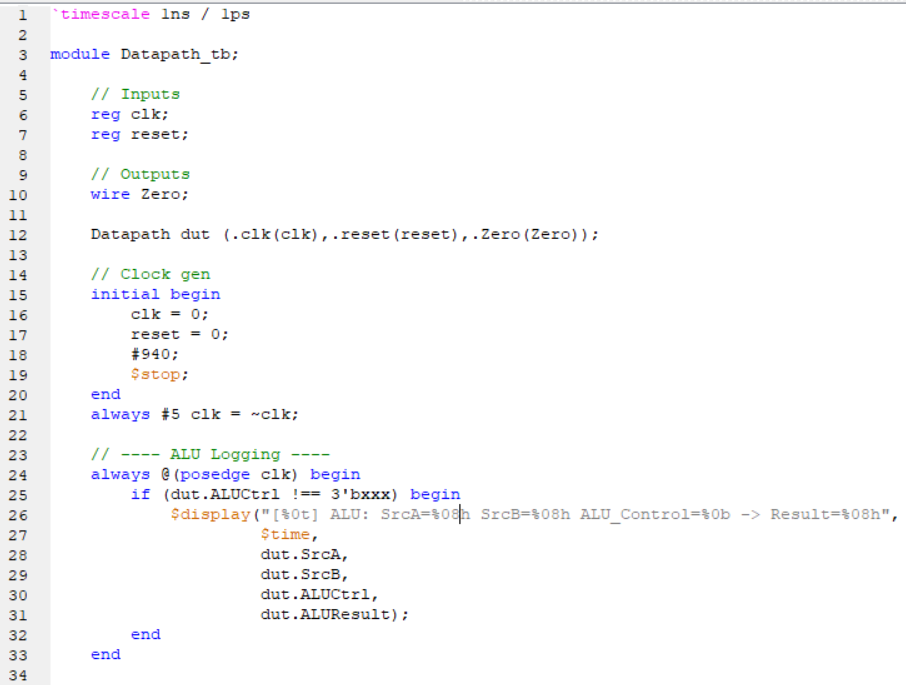
| 12 | j 14 | 000010 000000000000000000001110 |

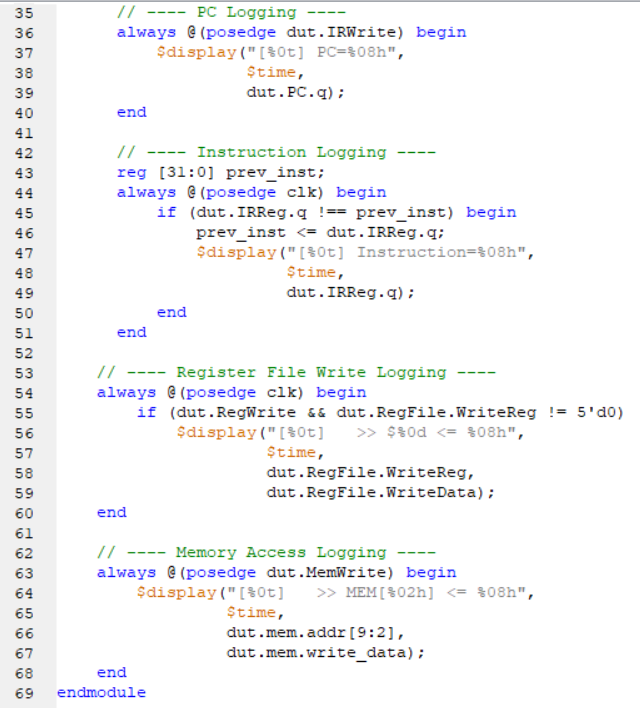
| 13 | addi $t0, $zero, 200 | 001000 00000 01000 0000000011001000 |

| 14 | addi $t1, $zero, 50 | 001000 00000 01001 0000000000110010 |

این فایل ورودی حافظه (Instruction Memory) است. هر خط دقیقاً معادل یک دستور ۳۲ بیتی است. در ماژول حافظه Verilog، با دستور $readmemb("init\_mem.data", mem) این فایل لود می‌شود.

**Datapath\_tb (Testbench)**





**این تست‌بنچ اجرای یک برنامه‌ی کامل روی ماژول Datapath را شبیه‌سازی می‌کند و با probe کردن سیگنال‌های داخلی (hierarchical references) لاگ‌های دقیقی از رفتار پردازنده چاپ می‌کند:**

* تکامل PC و واکشی دستور (IR)
* ورودی/خروجی و کنترل ALU
* نوشتن در Register File
* عملیات نوشتن در حافظه (Store)

این تست‌بنچ فرض می‌کند حافظه‌ی برنامه در ماژول Memory با فایل اولیه مثل (assembler/init\_mem.data) preload شده است.

## **Probe کردن سیگنال‌های داخلی:**

این تست‌بنچ از **مسیرهای سلسله‌مراتبی** استفاده می‌کند مانند (dut.ALUCtrl, dut.IRReg.q, dut.mem.addr) تا سیگنال‌ها/پورت‌های داخلی زیرماژول‌ها را مشاهده کند.

**نمونه‌ها:**

* **:dut.PC.q** خروجی رجیستر) PC نمونه‌ی Reg32 با پورت (q
* **:dut.IRReg.q** مقدار Instruction Register
* **:dut.RegFile.WriteReg/WriteData** پورت‌های ورودی زیرماژول RegisterFile
* **dut.mem.addr و** :**dut.mem.write\_data** پورت‌های زیرماژول Memory
* **:dut.SrcA, dut.SrcB, dut.ALUResult, dut.ALUCtrl** وایرهای داخلی datapath

## **بلاک‌های لاگ‌گیری:**

### ALU Logging(1:

 در هر **لبه‌ی بالارونده‌ی کلاک**، اگر ALUCtrl مقدار معتبر) غیر از (Xداشته باشد، ورودی‌ها/خروجی ALU لاگ می‌شوند.

 استفاده از **case-inequality** (!==) با 3'bxxx تضمین می‌کند فقط وقتی کنترل ALU مشخص است، چاپ انجام شود.

### PC Logging( 2در لحظه‌ی IRWrite:

 با **بالا رفتن**)  **IRWrite** معمولاً در (state FETCHمقدار فعلی PC چاپ می‌شود.

 از آن‌جا که IRWrite در ورود به FETCH یک پالس می‌گیرد، این چاپ معمولاً **PC پس از به‌روزرسانی (PC+4)** را نشان می‌دهد.

### Instruction Logging( 3با تشخیص تغییر:

 هر زمان مقدار IR عوض شود ) دستور جدید fetch شود (، یک خط لاگ چاپ می‌شود.

 رجیستر prev\_inst جلوی چاپ‌های تکراری را می‌گیرد.

### : Register File Write Logging(4

 در هر سیکل، اگر RegWrite=1 و مقصد صفر نباشد، عملیات **Write-Back** گزارش می‌شود:

* شماره رجیستر مقصد از پورت WriteReg
* داده‌ی نوشته‌شده از پورت WriteData

 این چاپ‌ها نشان می‌دهد که **کدام دستور** چه مقداری را در **کدام رجیستر** نوشته است (برای lw, addi, R-type و…

### : Memory Write Logging(5

 با **بالا رفتن MemWrite**، یک عمل Store گزارش می‌شود.

**dut.mem.addr[9:2]**  ایندکس word-aligned حافظه (عمق 256) را نشان می‌دهد.

**write\_data** محتوایی است که در آن آدرس نوشته می‌شود.

**خروجی رفتار** **پردازنده‌ی MIPS (Multicycle):**

[5000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=0 -> Result=00000000

[5000] Instruction=00000000

[5000] PC=00000000

[15000] ALU: SrcA=00000000 SrcB=00000004 ALU\_Control=10 -> Result=00000004

[25000] ALU: SrcA=00000004 SrcB=00000014 ALU\_Control=10 -> Result=00000018

[25000] Instruction=20080005

[35000] ALU: SrcA=00000000 SrcB=00000005 ALU\_Control=10 -> Result=00000005

[45000] ALU: SrcA=00000004 SrcB=00000000 ALU\_Control=10 -> Result=00000004

[45000] >> $8 <= 00000005

[45000] PC=00000004

[55000] ALU: SrcA=00000004 SrcB=00000004 ALU\_Control=10 -> Result=00000008

[65000] ALU: SrcA=00000008 SrcB=00000028 ALU\_Control=10 -> Result=00000030

[65000] Instruction=2009000a

[75000] ALU: SrcA=00000000 SrcB=0000000a ALU\_Control=10 -> Result=0000000a

[85000] ALU: SrcA=00000008 SrcB=00000000 ALU\_Control=10 -> Result=00000008

[85000] >> $9 <= 0000000a

[85000] PC=00000008

[95000] ALU: SrcA=00000008 SrcB=00000004 ALU\_Control=10 -> Result=0000000c

[105000] ALU: SrcA=0000000c SrcB=00014080 ALU\_Control=10 -> Result=0001408c

[105000] Instruction=01095020

[115000] ALU: SrcA=00000005 SrcB=0000000a ALU\_Control=10 -> Result=0000000f

[125000] ALU: SrcA=0000000c SrcB=0000000a ALU\_Control=10 -> Result=00000016

[125000] >> $10 <= 0000000f

[125000] PC=0000000c

[135000] ALU: SrcA=0000000c SrcB=00000004 ALU\_Control=10 -> Result=00000010

[145000] ALU: SrcA=00000010 SrcB=00016088 ALU\_Control=10 -> Result=00016098

[145000] Instruction=01285822

[155000] ALU: SrcA=0000000a SrcB=00000005 ALU\_Control=110 -> Result=00000005

[165000] ALU: SrcA=00000010 SrcB=00000005 ALU\_Control=10 -> Result=00000015

[165000] >> $11 <= 00000005

[165000] PC=00000010

[175000] ALU: SrcA=00000010 SrcB=00000004 ALU\_Control=10 -> Result=00000014

[185000] ALU: SrcA=00000014 SrcB=00018090 ALU\_Control=10 -> Result=000180a4

[185000] Instruction=01096024

[195000] ALU: SrcA=00000005 SrcB=0000000a ALU\_Control=0 -> Result=00000000

[205000] ALU: SrcA=00000014 SrcB=0000000a ALU\_Control=10 -> Result=0000001e

[205000] >> $12 <= 00000000

[205000] PC=00000014

[215000] ALU: SrcA=00000014 SrcB=00000004 ALU\_Control=10 -> Result=00000018

[225000] ALU: SrcA=00000018 SrcB=0001a094 ALU\_Control=10 -> Result=0001a0ac

[225000] Instruction=01096825

[235000] ALU: SrcA=00000005 SrcB=0000000a ALU\_Control=1 -> Result=0000000f

[245000] ALU: SrcA=00000018 SrcB=0000000a ALU\_Control=10 -> Result=00000022

[245000] >> $13 <= 0000000f

[245000] PC=00000018

[255000] ALU: SrcA=00000018 SrcB=00000004 ALU\_Control=10 -> Result=0000001c

[265000] ALU: SrcA=0000001c SrcB=0001c0a8 ALU\_Control=10 -> Result=0001c0c4

[265000] Instruction=0109702a

[275000] ALU: SrcA=00000005 SrcB=0000000a ALU\_Control=111 -> Result=00000001

[285000] ALU: SrcA=0000001c SrcB=0000000a ALU\_Control=10 -> Result=00000026

[285000] >> $14 <= 00000001

[285000] PC=0000001c

[295000] ALU: SrcA=0000001c SrcB=00000004 ALU\_Control=10 -> Result=00000020

[305000] ALU: SrcA=00000020 SrcB=00000000 ALU\_Control=10 -> Result=00000020

[305000] Instruction=ac0a0000

[315000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[315000] >> MEM[00] <= 0000000f

[325000] ALU: SrcA=00000020 SrcB=0000000f ALU\_Control=10 -> Result=0000002f

[325000] PC=00000020

[335000] ALU: SrcA=00000020 SrcB=00000004 ALU\_Control=10 -> Result=00000024

[345000] ALU: SrcA=00000024 SrcB=00000000 ALU\_Control=10 -> Result=00000024

[345000] Instruction=8c0f0000

[355000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[365000] ALU: SrcA=00000024 SrcB=00000000 ALU\_Control=10 -> Result=00000024

[375000] ALU: SrcA=00000024 SrcB=00000000 ALU\_Control=10 -> Result=00000024

[375000] >> $15 <= 0000000f

[375000] PC=00000024

[385000] ALU: SrcA=00000024 SrcB=00000004 ALU\_Control=10 -> Result=00000028

[395000] ALU: SrcA=00000028 SrcB=00000008 ALU\_Control=10 -> Result=00000030

[395000] Instruction=11ea0002

[405000] ALU: SrcA=0000000f SrcB=0000000f ALU\_Control=110 -> Result=00000000

[405000] PC=00000030

[415000] ALU: SrcA=00000030 SrcB=00000004 ALU\_Control=10 -> Result=00000034

[425000] ALU: SrcA=00000034 SrcB=00000038 ALU\_Control=10 -> Result=0000006c

[425000] Instruction=0800000e

[435000] ALU: SrcA=00000034 SrcB=00000000 ALU\_Control=10 -> Result=00000034

[435000] PC=00000038

[445000] ALU: SrcA=00000038 SrcB=00000004 ALU\_Control=10 -> Result=0000003c

[455000] ALU: SrcA=0000003c SrcB=000000c8 ALU\_Control=10 -> Result=00000104

[455000] Instruction=20090032

[465000] ALU: SrcA=00000000 SrcB=00000032 ALU\_Control=10 -> Result=00000032

[475000] ALU: SrcA=0000003c SrcB=0000000a ALU\_Control=10 -> Result=00000046

[475000] >> $9 <= 00000032

[475000] PC=0000003c

[485000] ALU: SrcA=0000003c SrcB=00000004 ALU\_Control=10 -> Result=00000040

[495000] ALU: SrcA=00000040 SrcB=00000000 ALU\_Control=10 -> Result=00000040

[495000] Instruction=00000000

[505000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[515000] ALU: SrcA=00000040 SrcB=00000000 ALU\_Control=10 -> Result=00000040

[515000] PC=00000040

[525000] ALU: SrcA=00000040 SrcB=00000004 ALU\_Control=10 -> Result=00000044

[535000] ALU: SrcA=00000044 SrcB=00000000 ALU\_Control=10 -> Result=00000044

[545000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[555000] ALU: SrcA=00000044 SrcB=00000000 ALU\_Control=10 -> Result=00000044

[555000] PC=00000044

[565000] ALU: SrcA=00000044 SrcB=00000004 ALU\_Control=10 -> Result=00000048

[575000] ALU: SrcA=00000048 SrcB=00000000 ALU\_Control=10 -> Result=00000048

[585000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[595000] ALU: SrcA=00000048 SrcB=00000000 ALU\_Control=10 -> Result=00000048

[595000] PC=00000048

[605000] ALU: SrcA=00000048 SrcB=00000004 ALU\_Control=10 -> Result=0000004c

[615000] ALU: SrcA=0000004c SrcB=00000000 ALU\_Control=10 -> Result=0000004c

[625000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[635000] ALU: SrcA=0000004c SrcB=00000000 ALU\_Control=10 -> Result=0000004c

[635000] PC=0000004c

[645000] ALU: SrcA=0000004c SrcB=00000004 ALU\_Control=10 -> Result=00000050

[655000] ALU: SrcA=00000050 SrcB=00000000 ALU\_Control=10 -> Result=00000050

[665000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[675000] ALU: SrcA=00000050 SrcB=00000000 ALU\_Control=10 -> Result=00000050

[675000] PC=00000050

[685000] ALU: SrcA=00000050 SrcB=00000004 ALU\_Control=10 -> Result=00000054

[695000] ALU: SrcA=00000054 SrcB=00000000 ALU\_Control=10 -> Result=00000054

[705000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[715000] ALU: SrcA=00000054 SrcB=00000000 ALU\_Control=10 -> Result=00000054

[715000] PC=00000054

[725000] ALU: SrcA=00000054 SrcB=00000004 ALU\_Control=10 -> Result=00000058

[735000] ALU: SrcA=00000058 SrcB=00000000 ALU\_Control=10 -> Result=00000058

[745000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[755000] ALU: SrcA=00000058 SrcB=00000000 ALU\_Control=10 -> Result=00000058

[755000] PC=00000058

[765000] ALU: SrcA=00000058 SrcB=00000004 ALU\_Control=10 -> Result=0000005c

[775000] ALU: SrcA=0000005c SrcB=00000000 ALU\_Control=10 -> Result=0000005c

[785000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[795000] ALU: SrcA=0000005c SrcB=00000000 ALU\_Control=10 -> Result=0000005c

[795000] PC=0000005c

[805000] ALU: SrcA=0000005c SrcB=00000004 ALU\_Control=10 -> Result=00000060

[815000] ALU: SrcA=00000060 SrcB=00000000 ALU\_Control=10 -> Result=00000060

[825000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[835000] ALU: SrcA=00000060 SrcB=00000000 ALU\_Control=10 -> Result=00000060

[835000] PC=00000060

[845000] ALU: SrcA=00000060 SrcB=00000004 ALU\_Control=10 -> Result=00000064

[855000] ALU: SrcA=00000064 SrcB=00000000 ALU\_Control=10 -> Result=00000064

[865000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[875000] ALU: SrcA=00000064 SrcB=00000000 ALU\_Control=10 -> Result=00000064

[875000] PC=00000064

[885000] ALU: SrcA=00000064 SrcB=00000004 ALU\_Control=10 -> Result=00000068

[895000] ALU: SrcA=00000068 SrcB=00000000 ALU\_Control=10 -> Result=00000068

[905000] ALU: SrcA=00000000 SrcB=00000000 ALU\_Control=10 -> Result=00000000

[915000] ALU: SrcA=00000068 SrcB=00000000 ALU\_Control=10 -> Result=00000068

[915000] PC=00000068

[925000] ALU: SrcA=00000068 SrcB=00000004 ALU\_Control=10 -> Result=0000006c

[935000] ALU: SrcA=0000006c SrcB=00000000 ALU\_Control=10 -> Result=0000006c

### Stopped at time : 940 ns

### # init\_mem.data → Assembly readable

### addi $t0, $zero, 1 # $t0 = 1

### addi $t1, $zero, 1 # $t1 = 1

### addi $t3, $zero, 2 # $t3 = 2

### addi $t4, $zero, 6 # $t4 = 6

### add $t2, $t0, $t1 # $t2 = $t0 + $t1

### addi $t3, $t3, 1 # $t3 = $t3 + 1

### beq $t3, $t4, 3 # if ($t3 == $t4) skip 3 instructions

### addi $t0, $t1, 0 # $t0 = $t1

### addi $t1, $t2, 0 # $t1 = $t2

### j 4 # jump to instruction at address 4

1. **واکشی و شروع اجرا**
   * در سیکل‌های ابتدایی مقدار اولیه‌ی PC برابر صفر است و ALU محاسبه‌ی PC + 4 را انجام می‌دهد.
   * اولین دستور خوانده‌شده 20080005 است که معادل دستور addi $t0, $zero, 5 می‌باشد.
2. **اجرای دستور ADDI**
   * ALU مقدار 0 (محتوای ($zero را با عدد ثابت 5 جمع می‌کند.
   * در مرحله‌ی WB مقدار 5 در ثبات ) $t0 یعنی ($8ذخیره می‌شود.
3. **اجرای دستور دوم (ADDI)**
   * دستور 2009000a معادل addi $t1, $zero, 10 واکشی می‌شود.
   * در پایان، مقدار 10 در ثبات ) $t1 یعنی ($9نوشته می‌شود.
4. **اجرای دستور R-type (ADD)**
   * دستور 01095020 معادل add $t2, $t0, $t1 است.
   * ALU عمل جمع 5 + 10 را انجام داده و نتیجه‌ی 15 در ثبات $t2 ذخیره می‌شود.
5. **اجرای دستور R-type (SUB)**
   * دستور 01285822 معادل sub $t3, $t1, $t0 است.
   * حاصل 10 - 5 = 5 در ثبات $t3 ذخیره می‌شود.
6. **اجرای دستور AND**
   * دستور 01096024 معادل and $t4, $t0, $t1 است.
   * نتیجه‌ی 5 AND 10 برابر صفر است و در $t4 ذخیره می‌شود.
7. **اجرای دستور OR**
   * دستور 01096825 معادل or $t5, $t0, $t1 است.
   * حاصل 5 OR 10 = 15 در $t5 نوشته می‌شود.
8. **اجرای دستور SLT**
   * دستور 0109702a معادل slt $t6, $t0, $t1 است.
   * چون 5 < 10، مقدار 1 در ثبات $t6 قرار می‌گیرد.
9. **اجرای دستور SW**
   * دستور ac0a0000 معادل sw $t2, 0($zero) است.
   * مقدار 15 (محتوای ($t2در آدرس حافظه 0 ذخیره می‌شود.
10. **اجرای دستور LW**
    * دستور 8c0f0000 معادل lw $t7, 0($zero) است.
    * مقدار 15 از حافظه خوانده شده و در ثبات $t7 قرار می‌گیرد.
11. **اجرای دستور شرطی BEQ**
    * دستور 11ea0002 معادل beq $t7, $t2, offset است.
    * چون مقدار $t7 و $t2 هر دو برابر 15 هستند، شرط برقرار شده و پرش انجام می‌شود PC به آدرس مقصد تغییر می‌کند.
12. **اجرای دستور JUMP**
    * دستور 0800000e معادل j 0x00000038 است.
    * PC به آدرس 0x38 جهش داده می‌شود و اجرای برنامه از آن نقطه ادامه پیدا می‌کند.
13. **پایان برنامه**
    * در نهایت چند دستور NOP (00000000) اجرا می‌شود و پردازنده در سیکل 940 ns متوقف می‌گردد.

## **جمع‌بندی**

خروجی شبیه‌سازی نشان داد که پردازنده‌ی MIPS چندچرخه طراحی‌شده قادر به اجرای صحیح انواع دستورها شامل:

* دستورهای Immediate مثل (ADDI)
* دستورهای R-type مثل ADD)، SUB، AND، OR، (SLT
* دستورهای حافظه‌ای LW)، (SW
* دستورهای کنترلی BEQ)، (JUMP

می‌باشد. همچنین ثبت تغییرات ثبات‌ها و حافظه در لاگ سیمولیشن تأییدکننده‌ی صحت عملکرد ماژول‌های **کنترل، ALU، رجیستر فایل و حافظه** است.