گزارش پروژه درس معماری کامپیوتر

استاد:
حمید سربازی آزاد

اعضای گروه:
مسیح بیگی‌ریزی
محمدامین کرمی
علیرضا حسین‌خانی
امین حسن‌زاده

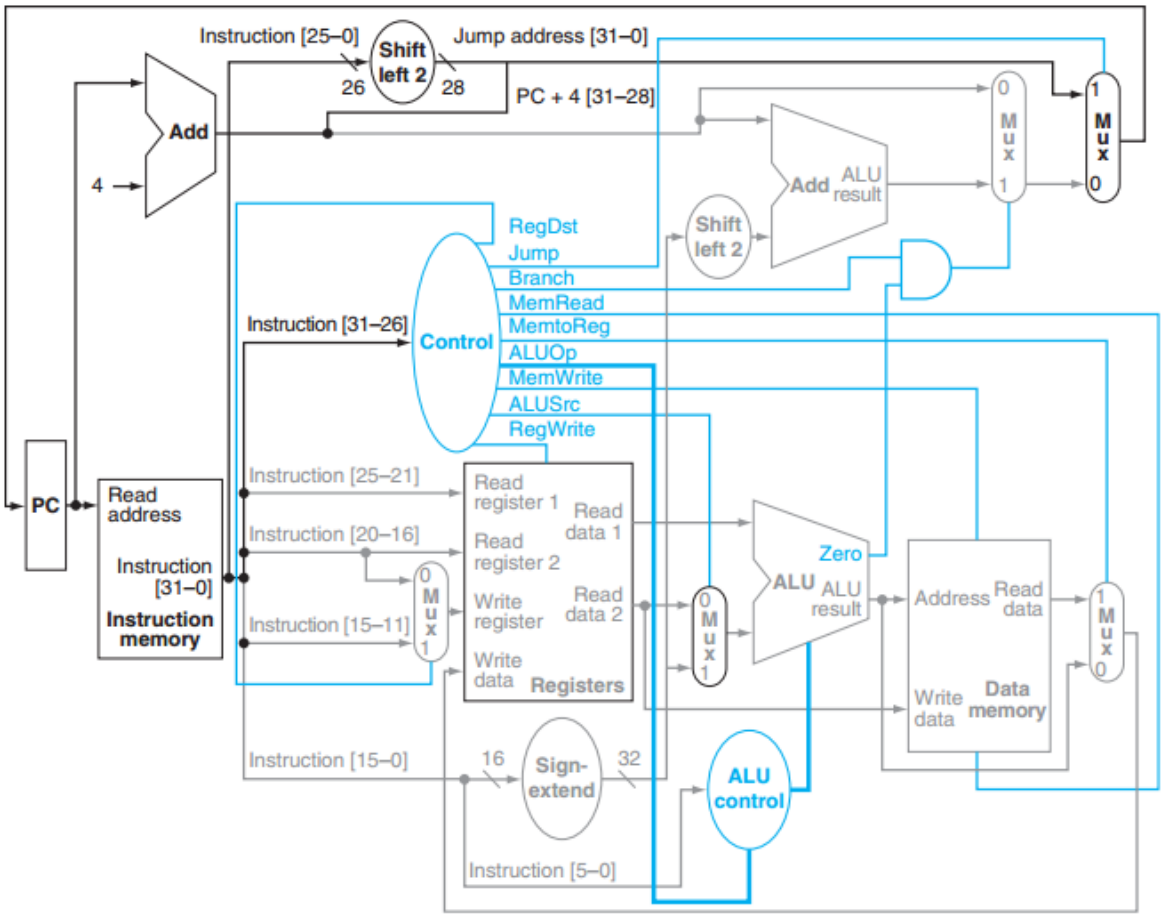
بهار ۱۴۰۱

فاز اول - پیاده‌سازی پردازنده mips به صورت single cycle

در این فاز با توجه به توضیحات داک پروژه اقدام به پیاده سازی پردازنده کردیم، در ادامه بخش‌های مختلف پروژه به تفکیک مورد بررسی قرار می‌گیرد:

Datapath

شمای کلی Datapath موجود در پردازنده به صورت زیر می‌باشد



ماژول های پیاده‌سازی شده

mips_core

این ماژول در عمل هسته‌ی مرکزی پردازنده‌ی ما می‌باشد که نمونه‌گیری از ماژول‌های regfile، pc_control، alu، control، alu_control و سیم‌کشی بین آن‌ها در آن انجام شده‌است، در واقع وظیفه اصلی این ماژول برقراری ارتباط بین ماژول‌ها با یکدیگر و misp_machine است.

alu

این ماژول وظیفه‌ی انجام محاسبات بر روی دو ورودی داده شده بر اساس سیگنال کنترلی‌ای که از alu_control دریافت می‌کند را برعهده دارد. خروجی این ماژول یک عدد ۳۲ بیتی به نام alu_r و یک سیگنال تک‌بیتی به نام zero (هنگامی فعال می‌شود که مقدار alu_result برابر با 0 باشد) می‌باشد.

alu_control

وظیفه‌ی این ماژول تولید سیگنال کنترلی control برای ماژول alu می‌باشد. در دستورات سری R این سیگنال به کمک func که بخشی از دستور است تعیین می‌شود و در سایر موارد به کمک سیگنال alu_op که از ماژول control خارج شده است تعیین می‌گردد.

control

وظیفه این ماژول تولید سیگنال‌های کنترلی برای سایر ماژول‌ها بر اساس opcode و func موجود در instruction است که به روش hardwired و direct method پیاده‌سازی شده است.

pc_control

در این ماژول بر اساس سیگنال‌های کنترلی برنچ‌ها و جامپ‌ها و همچنین مقادیر آدرس و ورودی immidiate مقدار بعدی PC تعیین می‌شود. وظیفه این ماژول جداسازی پیچیدگی‌های برنچینگ از کد اصلی است.

خروجی تست‌ها

```
mohamadamin@Mohamadamin:~/projects/CA/project-comma$ make verify-all |grep 'Cycle\|diff -u test/default\|All'
=== Simulation Cycle 9 ===
diff -u test/default/brtest0.reg output/regdump.reg 1>&2
=== Simulation Cycle 6 ===
diff -u test/default/addiu.reg output/regdump.reg 1>&2
=== Simulation Cycle 8 ===
diff -u test/default/shifttest.reg output/regdump.reg 1>&2
=== Simulation Cycle 5 ===
diff -u test/default/brtest2.reg output/regdump.reg 1>&2
=== Simulation Cycle 16 ===
diff -u test/default/arithtest.reg output/regdump.reg 1>&2
=== Simulation Cycle 31 ===
diff -u test/default/memtest0.reg output/regdump.reg 1>&2
=== Simulation Cycle 5 ===
diff -u test/default/addtest.reg output/regdump.reg 1>&2
All tests passed! (7 tests)
```

فاز دوم - پیاده‌سازی حافظه‌ی نهان (cache)

در این فاز در ابتدا ماژول mips_core را به گونه‌ای تغییر دادیم که امکان دریافت و ذخیره‌ی داده در مموری با تاخیر را انجام دهد. انجام دستورات مرتبط به مموری ۴ سایکل تاخیر دارد.

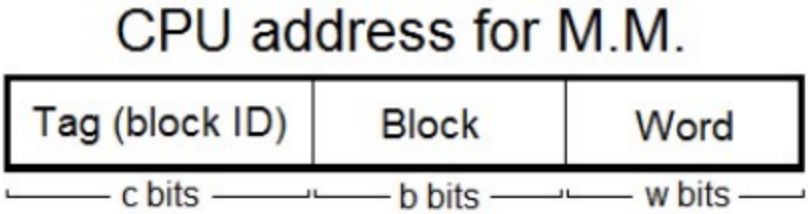
خروجی تست ها پس از این تغییر به صورت زیر شد:

```
mohamadamin@Mohamadamin:~/projects/CA/project-comma$ make verify-all |grep 'Cycle\|diff -u test/default\|All'
=== Simulation Cycle 9 ===
diff -u test/default/brtest0.reg output/regdump.reg 1>&2
=== Simulation Cycle 84 ===
diff -u test/default/sb.reg output/regdump.reg 1>&2
=== Simulation Cycle 6 ===
diff -u test/default/addiu.reg output/regdump.reg 1>&2
=== Simulation Cycle 8 ===
diff -u test/default/shifttest.reg output/regdump.reg 1>&2
=== Simulation Cycle 5 ===
diff -u test/default/brtest2.reg output/regdump.reg 1>&2
=== Simulation Cycle 16 ===
diff -u test/default/arithtest.reg output/regdump.reg 1>&2
=== Simulation Cycle 95 ===
diff -u test/default/memtest0.reg output/regdump.reg 1>&2
=== Simulation Cycle 5 ===
diff -u test/default/addtest.reg output/regdump.reg 1>&2
All tests passed! (8 tests)
```

از آنجا که به ازای هر دستور مموری باید ۴ کلاک صبر کنیم تا داده به‌دست آید یا داده از مموری خوانده شود، نیازمند کشی هستیم تا این تاخیر را کاهش دهد.

برای این کار یک Direct-Mapped Cache با مشخصات زیر در ماژول cache.sv پیاده سازی شد:

- Cache size = 8 Kbytes = 2^{13} bytes = 2^{11} blocks
- Block size = 1 word = 32 bits = 4 bytes
- Replacement policy = Direct-Mapped
- Write scheme = Write-back scheme



c = 19 bits ([31:13]) b = 11 bits ([13:2]) w = 0 bit

علت آن که هر بلاک را برابر با یک کلمه گرفتیم، این است که در هر بار خواندن مموری تنها دسترسی به یک کلمه داریم و اگر طول هز بلاک بیشتر از یک کلمه می‌بود، مجبور به چند بار خواندن از مموری در مواقع miss و به طبع آن افزایش کلاک بودیم.

علت آن که از سیاست جایگزینی direct-mapped استفاده کردیم، کاربردی بودن آن در این حجم از کش و همچنین سادگی پیاده‌سازی آن بود.

نحوه برقراری ارتباط کش با مموری

در ابتدا تمام اطلاعات موجود در کش مقدار نادرستی دارند پس valid bit در تمام بلاک ها برابر با 0 می‌باشد.

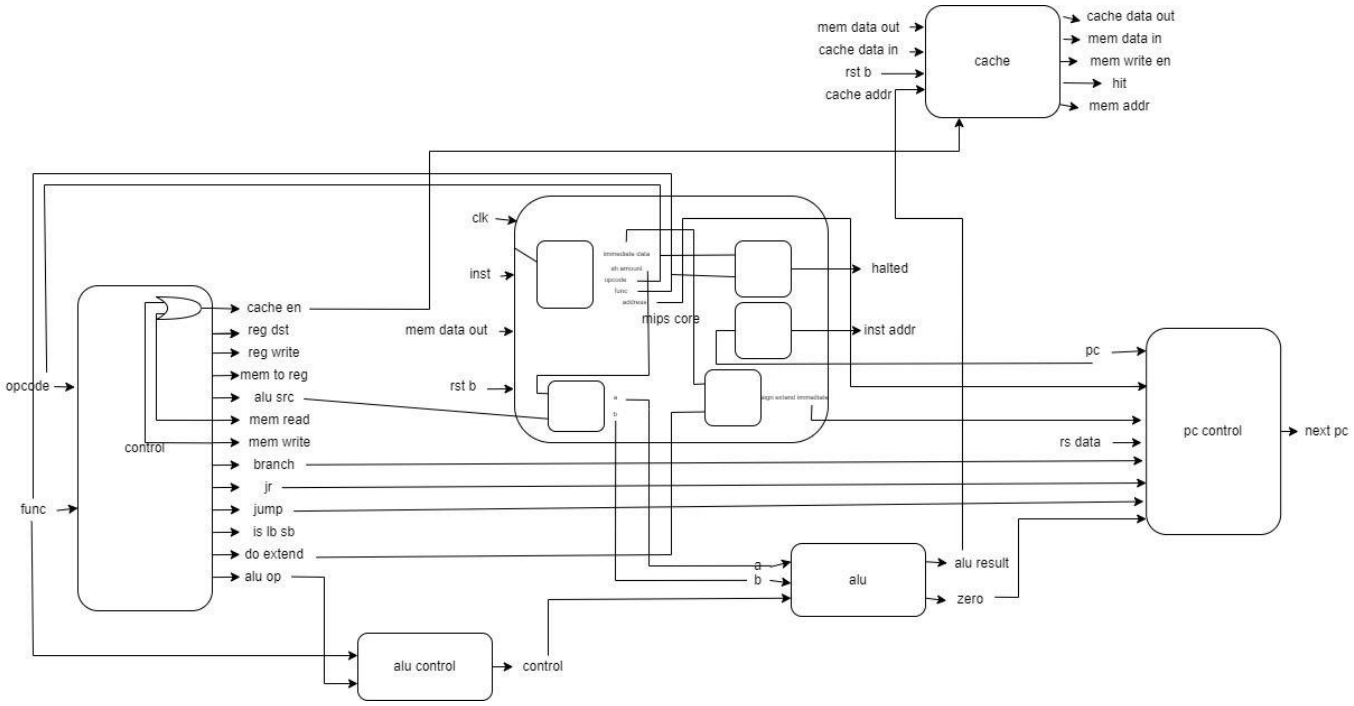
هنگامی که درخواست برای خواندن از کش داده شود، اگر valid bit برابر با 0 باشد یا tag یا هیچکدام از بلاک ها برابر نشود، آن داده از حافظه پس از ۴ کلاک خوانده شده و بر روی بلاکی از کش قرار می‌گیرد.

هنگامی که درخواست برای نوشتن بر روی حافظه داریم، در صورتی که آدرس مورد نظر در حافظه کش وجود داشت، داده‌ی جدید بر روی کش تغییر می‌کند و مقدار dirty bit مربوط به آن بلاک برابر با ۱ می‌شود. در صورتی که آدرس مورد نظر در حافظه کش وجود نداشت، در ابتدا پس از ۴ کلاک داده از مموری به کش انتقال می‌یابد و سپس داده‌ی جدید جایگزین می‌گردد.

هنگامی که حافظه‌ای از مموری خوانده می‌شود، اگر مقدار قبلی موجود در کش مقدار valid ای باشد و dirty bit آن برابر با ۱ باشد در ابتدا این بلاک پس از ۴ کلاک بر روی حافظه‌ی اصلی نوشته می‌شود و سپس مقدار جدید جایگزین می‌گردد.

Datapath

پس از اضافه شدن کش، datapath به صورت زیر خواهد بود:



خروجی تست‌ها

با استفاده از این کش، پس از ران کردن تست‌ها مقادیر سایکل تست‌های مربوط به ممووری به طور چشمگیری کاهش می‌یابد:

```
mohamadamin@Mohamadamin:~/projects/CA/project-comma$ make verify-all |grep 'Cycle\|diff -u test/default\|All'
=== Simulation Cycle 9 ===
diff -u test/default/brtest0.reg output/regdump.reg 1>&2
=== Simulation Cycle 32 ===
diff -u test/default/sb.reg output/regdump.reg 1>&2
=== Simulation Cycle 6 ===
diff -u test/default/addiu.reg output/regdump.reg 1>&2
=== Simulation Cycle 8 ===
diff -u test/default/shifttest.reg output/regdump.reg 1>&2
=== Simulation Cycle 5 ===
diff -u test/default/brtest2.reg output/regdump.reg 1>&2
=== Simulation Cycle 16 ===
diff -u test/default/arithtest.reg output/regdump.reg 1>&2
=== Simulation Cycle 51 ===
diff -u test/default/memtest0.reg output/regdump.reg 1>&2
=== Simulation Cycle 5 ===
diff -u test/default/addtest.reg output/regdump.reg 1>&2
All tests passed! (8 tests)
```

فاز سوم - پیاده‌سازی پردازنده به صورت خط لوله(pipline)

در این فاز خط لوله ای با ۵ stage طراحی و پیاده سازی کردیم که استیج ها در ادامه آمده است. علت استفاده از پایپلاین به طول ۵ این است که تقریبا در تمام منابع موجود پردازنده با ۵ استیج پایپلاین پیاده‌سازی شده است، همچنین با ۵ مرحله گرفتن و نه ۳ مرحله گرفتن سرعت کلاک بیشتر خواهد شد و باعث سریع‌تر شدن تا ۵ برابر حالت عادی خواهد شد(در صورتی که پایپلاین کامل پر باشد).

IF (Instruction Fetch)

در این stage دستور از مموری خوانده شده و PC ست شده و توسط رجیستر IF_to_ID داده‌های مورد نیاز به stage بعدی منتقل می‌شوند.

همچنین در این مرحله سیگنال flush توسط pc_control تعیین می‌گردد تا در مواقعی که دستورات branch وارد پایپلاین شده و دستور بعد از آن نباید اجرا شود، جلوی پیش‌روی این دستور در استیج‌های بعد را بگیرد، بدین صورت که با پاس دادن سیگنال flush به رجیستر IF_to_ID جلوی انتشار دستور اشتباه گرفته می‌شود و این دستور به مرحله ID و مراحل دیگر نخواهد رسید.

ID (Instruction Decode)

در این stage دستوری که در stage قبلی فچ شده توسط کنترلر decode می‌شود و سیگنال های کنترلی تشکیل شده توسط رجیستر ID_to_EXE به مرحله execute منتقل می‌شوند.
علاوه بر این موارد در این stage موارد forwarding نیز هندل شده است (در ادامه بررسی خواهد شد).

EXE (Execution of Instruction)

در این stage دستوری که در stage قبلی decode شده توسط ALU اجرا می‌شود و داده خروجی تولید می‌شود. داده خروجی و سیگنال های دیگر توسط رجیستر EXE_to_MEM به stage بعدی منتقل می‌شوند.

MEM (Mem write-read)

در این stage با توجه به سیگنال‌های ورودی داده ها از cache خوانده و یا روی cache نوشته خواهند شد. همچنین در این مرحله از آنجا که دسترسی به مموری با ۴ کلاک تاخیر همراه است، سیگنال کنترلی freeze تولید شده که وظیفه آن توقف خط لوله و جلوگیری از fetch کردن دستور جدید تا زمانی که داده از مموری خوانده شود است. با پاس دادن این سیگنال به ماژول‌های EXE_to_MEM، ID_to_EXE، IF_to_ID، IF_stage از پیش‌روی pipeline جلوگیری می‌شود.

WB (Write back to register)

در این stage بر اساس سیستم های کنترلی که توسط رجیسترها تا این مرحله منتقل شده اند، عمل write back روی رجیستر انجام می‌گیرد.

همچنین برای رفع control dependency و data dependency دو ماژول hazard_detector و forwarding را پیاده‌سازی کردیم. که در ادامه به بررسی آنها می‌پردازیم:

hazard_detector

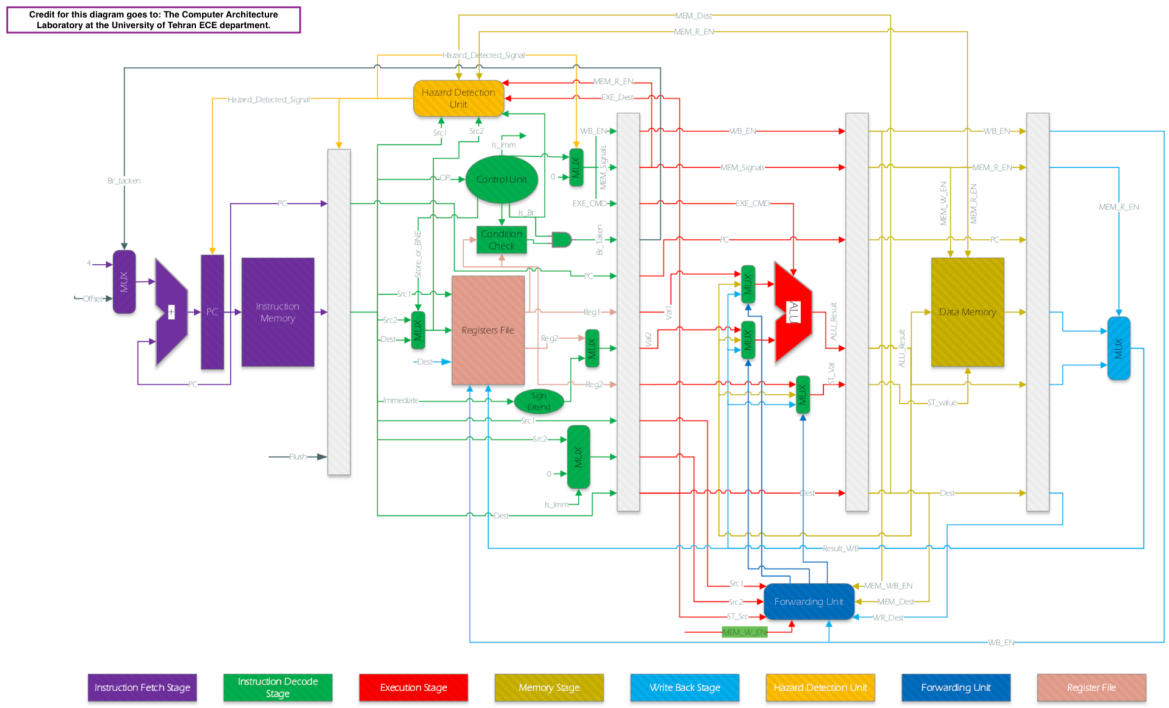
در این ماژول با گرفتن ورودی‌های شماره رجیسترها در مرحله ID، رجیسترِ مقصد و سیگنال reg_write (اگر فعال باشد نوشتن روی رجیستر صورت می‌گیرد) در مراحل EXE، MEM و WB، سیگنال های کنترلی is_reg1_valid و is_reg2_valid در مرحله ID (که در صورت یک بودن مشخص می‌کنند که دیتای آن رجیستر مورد استفاده قرار می‌گیرد) اقدام به تشخیص مخاطره وابستگی داده‌ها در استیج‌های مختلف و می‌کند.

forwarding

این ماژول بر اساس خروجی‌های hazard_detector و مقادیر موجود در استیج‌های EXE, MEM, WB داده‌ی صحیح مورد استفاده در مرحله EXE را تشخیص می‌دهد، بدین صورت که اگر آن رجیستر هر هیچ مرحله‌ای هازارد نداشته باشد، مقدار موجود در regfile، اگر در یک مرحله هازارد وجود داشت، مقدار موجود در آن مرحله انتخاب می‌گردد.

Datapath

در پیاده‌سازی این فاز از شکل زیر الهام گرفته شده است:



خروجی تست‌ها

پس از ایجاد پایپلاین خروجی تست‌ها به صورت زیر می‌باشد:

```
mohamadamin@Mohamadamin:~/projects/CA/project-comma$ make verify-all |grep 'Cycle\|diff -u test/default\|All
=== Simulation Cycle 15 ===
diff -u test/default/brtest0.reg output/regdump.reg 1>&2
=== Simulation Cycle 36 ===
diff -u test/default/sb.reg output/regdump.reg 1>&2
=== Simulation Cycle 10 ===
diff -u test/default/addiu.reg output/regdump.reg 1>&2
=== Simulation Cycle 12 ===
diff -u test/default/shifttest.reg output/regdump.reg 1>&2
=== Simulation Cycle 11 ===
diff -u test/default/brtest2.reg output/regdump.reg 1>&2
=== Simulation Cycle 20 ===
diff -u test/default/arithtest.reg output/regdump.reg 1>&2
=== Simulation Cycle 55 ===
diff -u test/default/memtest0.reg output/regdump.reg 1>&2
=== Simulation Cycle 9 ===
diff -u test/default/addtest.reg output/regdump.reg 1>&2
All tests passed! (8 tests)
```

فاز چهارم - افزودن کمک‌پردازنده اعداد ممیز شناور به پردازنده

در این فاز دستوراتی را جهت پردازش اعداد ممیز شناور به دستورات اصلی پردازنده اضافه شده و سپس به کمک دستورات جدید افزوده شده دو برنامه که در داک خواسته شده بود نوشته شده است. دقت شود که این دو برنامه به صورت دستی به کد ماشین ترجمه شده‌اند چرا که ابزاری جهت کامپایل برنامه‌ی اسمبلی به کد ماشین نداشتیم. شرح دستورات جدید و خروجی برنامه‌ها در زیر آمده است. همچنین سیگنال‌های خطای DBZ, QNAN, SNAN, Underflow, Overflow نیز توسط ماژول floating_point_alu تولید می‌شود.

شرح دستورات جدید

طراحی قالب دستورات براساس دستورات فرمت R پردازنده اصلی بوده و آپکد همه‌ی دستورات جدید 000000 می‌باشد و مشابه دستورات سری R با توجه به فیلد Func دستورات از هم تمایز داده می‌شوند. در زیر تصویر این قالب آمده است. چون پیاده سازی دستور شیفت ممیز شناور مطلوب ما نبوده فیلد Sh.Amount همواره مقدار صفر دارد یا به بیان بهتر مقدار این فیلد اهمیتی ندارد.

Opcode	$rs	$rt	$rd	Sh.Amount	Func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

1. **ADDF (Func = 111111)**

   این دستور دو عدد ممیز شناور داخل ثبات‌های rs, rt را با هم جمع می‌کند و حاصل را در rd ذخیره می‌کند.

2. **SUBF (Func = 111110)**

   این دستور عدد ممیز شناور داخل ثبات rt را از rs کم می‌کند و حاصل را در ثبات rd ذخیره می‌کند.

3. **MULTF (Func = 111101)**

   این دستور دو عدد ممیز شناور داخل ثبات‌های rt, rs را در هم ضرب می‌کند و حاصل را در ثبات rd ذخیره می‌کند.

4. **GF (Func = 111011)**

   این دستور دو عدد ممیز شناور داخل ثبات‌های rs, rt را با هم مقایسه می‌کند و عددی که بزرگتر است را در rd ذخیره می‌کند.

5. **DIVF (Func = 110111)**

   این دستور عدد ممیز شناور در ثبات rs را بر عدد ممیز شناور ثبات rt تقسیم می‌کند و حاصل را در ثبات rd ذخیره می‌کند.

6. **INVF (Func = 101111)**

   این دستور معکوس عدد ممیز شناور در ثبات rs را در ثبات rd ذخیره می‌کند

7. **ROUNDF (Func = 011111)**

این دستور مقدار رندشده‌ی عدد ممیز شناور داخل ثبات rs را در ثبات rd ذخیره می‌کند.

8. LF (Func = 001111)

این دستور مقدار ممیز شناور در ثبات‌های rs, rt را با هم مقایسه می‌کند و مقدار کوچکتر را در ثبات rd ذخیره می‌کند.

شرح خروجی برنامه‌های نوشته شده

برنامه اول مربوط به تست دستورات است که خروجی هر دستور در یکی از ثبات‌ها نوشته شده است. این برنامه با نام **temp.mem** در پوشه تست‌ها موجود است. جهت سهولت در خواندن محتویات ثبات‌ها به صورت ممیزشناور نمایش داده شده‌اند.

R10 = R3 + R9 (ADDF)

R11 = R3 - R9 (SUBF)

R12 = R3 * R9 (MULTF)

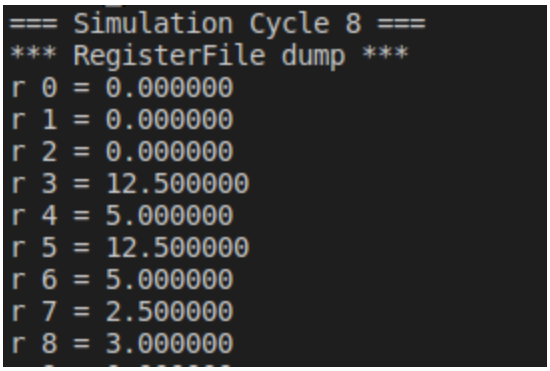
R13 = R3 if R3 > R9 else R9 (GF)

R14 = R3 / R9 (DIVF)

R15 = round(R3) (ROUNDF)

```
=== Simulation Cycle 11 ===
*** RegisterFile dump ***
r 0 = 0.000000
r 1 = 0.000000
r 2 = 0.000000
r 3 = 2.500000
r 4 = 0.000000
r 5 = 0.000000
r 6 = 0.000000
r 7 = 0.000000
r 8 = 0.000000
r 9 = 1.250000
r10 = 3.750000
r11 = 1.250000
r12 = 3.125000
r13 = 2.500000
r14 = 0.400000
r15 = 3.000000
```

برنامه دوم دو عدد داخل ثبات‌های R3, R4 را با هم مقایسه می‌کند و هر کدام که بزرگتر بود را بر دیگری تقسیم می‌کند و حاصل را رند می‌کند و در ثبات R8 ذخیره می‌کند. این برنامه با نام **program.mem** در پوشه تست‌ها موجود است. جهت سهولت در خواندن محتویات ثبات‌ها به صورت ممیز شناور نمایش داده شده‌اند.



دقت شود که کد ماشین هر دو برنامه به صورت دستی و با توجه به فرمت دستورات جدید مربوط به اعداد ممیز شناور که طراحی و پیاده‌سازی شده‌اند نوشته شده است.