



پیاده سازی پردازنده ARM

گردآورندگان:

دکتر علیرضا یزدان پناه

مهندس ادریس نصیحت کن

مهندس مرضیه رستگار

اهداف

- ۱- یادگیری مفاهیم اصلی معماری کامپیوتر
- ۲- یادگیری مفاهیم خط لوله در پردازنده
- ۳- تأثیرات اجزای مختلف پردازنده در کارایی آن و نحوه افزایش آن
- ۴- یادگیری طراحی سخت افزار
- ۵- نحوه کدنویسی Verilog با قابلیت سنتز
- ۶- نحوه عیب یابی و تست مدارهای سخت افزاری طراحی شده

توضیحات کلی

- ۱- در این آزمایش باید یک پردازنده ARM ساده که دارای ۱۳ دستور العمل اصلی است، پیاده سازی گردد.
- ۲- معماری اصلی این پردازنده را طراحی و کد Verilog آن را (با توضیحات کامل) به طور سنتز شدنی نوشته شود.
- ۳- ابتدا کد را با استفاده از ModelSim شبیه سازی و نتایج آن را در آزمایشگاه نشان دهید. سپس کد را با استفاده از Quartus II سنتز کنید (نتایج سنتز باید در گزارش کار بیاید) و سپس برد را برنامه ریزی کنید (نتایج اجرای برنامه بر روی برد باید در گزارش کار بیاید).
- ۴- برای هر قسمت از این پردازنده (هر ماژول) باید یک ماژول تست نوشته و آن را شبیه سازی و تست نمایید.
- ۵- پس از طراحی تمامی ماژول های پردازنده، ماژول ها را به یکدیگر متصل نمایید و کل پردازنده را شبیه سازی و تست نمایید.
- ۶- برای تست نهایی پردازنده یک ماژول سطح بالا (Testbench) طراحی کنید که کد دودویی یک عملیات (مانند حاصل جمع دو عدد) را داخل Program ROM یا قرار دهید و آن را خط به خط اجرا نمایید. تا در نهایت جواب نهایی حاصل شود.

توضیحات پیاده سازی به صورت مجازی

- ۱- پردازنده به صورت کامل بر روی نرم افزار ModelSim پیاده سازی شود و زمان اجرا در آن محاسبه شود.



- ۲- برای گرفتن گزارش های سخت افزاری پردازنده بر روی نرم افزار Quartus II ستر کنید و گزارش های سخت افزاری مانند تعداد امان های منطقی و ... را براساس آن گزارش کنید. (مدل FPGA مورد استفاده را در گزارش ذکر کنید)
- ۳- مباحث مربوط به پیاده سازی روی برد در شرایط مجازی حذف شده است.

مقدمه

ARM نوعی از معماری پردازنده های کامپیوتری است که بر طبق طراحی RISC CPU و توسط کمپانی بریتانیایی ARM Holding طراحی شده است. معماری ARM که دستورالعمل های ۳۲ بیتی را پردازش می کند از دهه ۱۹۸۰ تا به امروز در حال توسعه است.

شرکت ARM Holding تولیدکننده پردازنده نیست و در عوض گواهی استفاده از معماری ARM را به دیگر تولیدکنندگان نیمه هادی می فروشد. کمپانی ها نیز به راحتی تراشه های خود را براساس معماری ARM تولید می کنند. از جمله شرکت هایی که پردازنده خود را براساس معماری ARM طراحی می کنند می توان به اپل^۱ در تراشه های Ax، سامسونگ^۲ در پردازنده های Exynos، انویدیا^۳ در تگرا^۴ و کوالکام^۵ در پردازنده های Snapdragon اشاره کرد.

تا سال ۲۰۱۷ بیش از ۱۰۰ میلیارد پردازنده ARM در جهان مورد استفاده قرار گرفته است. در ۴ سال منتهی به ۲۰۱۷، ۲۲ میلیارد از این پردازنده ها در گوشی های همراه، ۱۸ میلیارد در سیستم های نهفته، ۷ میلیارد در مصارف صنعتی و ۳ میلیارد در دستگاه های خانگی مورد استفاده قرار گرفته است.

در جدول ۱ نسل های مختلف معماری ARM لیست شده است. معماری ARMv1 که ساده ترین پردازنده ارائه شده توسط این شرکت است از نوع in-order و دارای یک خط لوله^۶ ۳ مرحله ای شامل واکنشی دستور^۷، کدگذاری دستور^۸ و اجرا^۹ است. در نسل های مختلف پردازنده ARM براساس میزان کارایی مورد نیاز معماری های متفاوتی ارائه شده است که برای مطالعه بیشتر می توانید به لینک <http://infocenter.arm.com/help/index.jsp> مراجعه کنید.

Instruction set Architecture	Core bit-width	Cores		Profile
		Arm Holdings	Third-party	
ARMv1	32[a 1]	ARM1		Classic

^۱Apple

^۲Samsung

^۳Nvidia

^۴Tegra

^۵Qualcomm

^۶Pipeline

^۷Instruction Fetch (IF)

^۸Instruction Decode (ID)

^۹Execute



دستور کار آزمایشگاه معماری کامپیوتر
بخش سخت افزار، دانشکده برق و کامپیوتر، دانشگاه تهران
آزمایش اول: پیاده سازی پردازنده ARM



ARMv2	32[a 1]	ARM2, ARM250, ARM3	Amber, STORM Open Soft Core[40]	Classic
ARMv3	32[a 2]	ARM6, ARM7		Classic
ARMv4	32[a 2]	ARM8	StrongARM, FA526, ZAP Open Source Processor Core[41]	Classic
ARMv4T	32[a 2]	ARM7TDMI, ARM9TDMI, SecurCore SC100		Classic
ARMv5TE	32	ARM7EJ, ARM9E, ARM10E	XScale, FA626TE, Feroceon, PJ1/Mohawk	Classic
ARMv6	32	ARM11		Classic
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000		Microcontroller
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300		Microcontroller
ARMv7E-M	32	ARM Cortex-M4, ARM Cortex-M7		Microcontroller
ARMv8-M	32	ARM Cortex-M23,[42] ARM Cortex-M33[43]		Microcontroller
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8		Real-time
ARMv8-R	32	ARM Cortex-R52		Real-time
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Qualcomm Krait/Scorpion, PJ4/Sheeva, Apple Swift	Application
ARMv8-A	32	ARM Cortex-A32		Application
	64/32	ARM Cortex-A35,[48] ARM Cortex-A53, ARM Cortex-A57,[49] ARM Cortex-A72,[50] ARM Cortex-A73[51]	X-Gene, Nvidia Project Denver 1/2, Cavium Thunder X[52][53][54], AMD K12, Apple Cyclone/Typhoon/Twister/Hurricane/Zephyr/Monsoon/Mistral, Qualcomm Kryo, Samsung M1/M2/M3 ("Mongoose")[55]	Application
ARMv8.1-A	64/32	TBA	ThunderX2[58]	Application
ARMv8.2-A	64/32	ARM Cortex-A55,[59] ARM Cortex-A75,[60] ARM Cortex-A76[61], Cortex-A65, Neoverse E1, Neoverse E1	Nvidia Carmel, Samsung M4[62]	Application



ARMv8.3-A	64/32	TBA	Apple Vortex/Tempest	Application
ARMv8.4-A	64/32	TBA		Application
ARMv8.5-A	64/32	TBA		Application

جدول ۱- لیست خانواده های پردازنده ی ARM

برای پیاده سازی در درس آزمایشگاه معماری کامپیوتر دانشگاه تهران معماری پردازنده ARM968E-S از خانواده ARM9 و ARM9E با هدف مروری بر مطالب ارائه شده در درس معماری کامپیوتر، آشنایی با یک پردازنده جدید و ISA^{۱۰} متفاوت انتخاب شده است. خانواده پردازنده ARM9 و ARM9E از سال ۱۹۹۸ تا ۲۰۰۶ به بازار عرضه شد و شامل پردازنده های ARM926EJ-S، ARM946E-S، ARM966E-S، ARM968E-S، ARM996HS، ARM920T و ARM922T می شود. پردازنده ARM968E-S در سال ۲۰۰۴ مبتنی بر معماری دستورات ARMv5TE ارائه شد. این پردازنده از نوع in-order بوده و دارای خط لوله ۵ مرحله ای و از لحاظ معماری شباهت زیادی به معماری MIPS پایه دارد.

مشخصات پردازنده

- ۱- پهنای خط داده: ۳۲ بیت
- ۲- تعداد مراحل خط لوله: ۵ مرحله ای
- ۳- تعداد دستورات: ۱۳ دستور
- ۴- میزان تاخیر انشعاب: ۲ مرحله
- ۵- ۱۶ ثبات همه منظوره (ثبات ۱۵ به منظور PC استفاده می شود و ثبات ۱۴ نیز به عنوان Link Register)
- ۶- آدرس دهی برحسب بایت و فضای آدرس دستورات (Instructions) و داده (Data) تفکیک شده می باشد.
(آدرس ۰ تا ۱۰۲۳ به Program ROM اختصاص دارد و آدرس ۱۰۲۴ به بعد به RAM تعلق دارد.)
- ۷- تمامی پرش ها از نوع محلی تعریف شده است و پس از پرش مقدار رجیستر شمارنده دستور به شکل زیر خواهد بود.
$$PC = PC + (\text{signed_immed_24} \ll 2) + 4$$
- ۸- قابلیت تشخیص و جلوگیری هازاد داده ای (Hazard Detection Unit) دارد و واحد ارسال به جلو (Forwarding Unit) ندارد.

¹⁰Instruction Set Architecture



مجموعه دستورات پردازنده

پردازنده‌ای که در این آزمایش طراحی و پیاده سازی می گردد، یک پردازنده ARM ساده شده است که دارای ۱۲ دستور العمل اصلی است. این پردازنده قابلیت انجام عملیات های ریاضی (ADD, ADC, SUB, SBC)، عملیات های منطقی (AND, ORR, EOR)، عملیات های مقایسه (CMP, TST)، عملیات خواندن و نوشتن در حافظه (LD, ST)، عملیات پرش (B) را دارد. لیست عملیات ها به همراه جزئیات آنها در جدول ۲ آورده شده است. دستور NOP به عنوان یک دستور پیاده سازی نمی شود.

R-type Instructions		Description	Bits							
			31:28	27:26	25	24:21	20	19:16	15:12	11:00
			Cond.	Mode	I	OP-Code	S	Rn	Rd	shifter operand
0	NOP ^{۱۱}	No Operation	1110	00	0	0000	0	0000	0000	000000000000
1	MOV	Move	cond	00	1	1101	S	0000	Rd	shifter operand
2	MVN ^{۱۲}	Move NOT	cond	00	1	1111	S	0000	Rd	shifter operand
3	ADD	Add	cond	00	1	0100	S	Rn	Rd	shifter operand
4	ADC	Add with Carry	cond	00	1	0101	S	Rn	Rd	shifter operand
5	SUB	Subtraction	cond	00	1	0010	S	Rn	Rd	shifter operand
6	SBC	Subtract with Carry	cond	00	1	0110	S	Rn	Rd	shifter operand
7	AND	And	cond	00	1	0000	S	Rn	Rd	shifter operand
8	ORR	Or	cond	00	1	1100	S	Rn	Rd	shifter operand
9	EOR	Exclusive OR	cond	00	1	0001	S	Rn	Rd	shifter operand
10	CMP	Compare	cond	00	1	1010	1	Rn	0000	shifter operand
11	TST ^{۱۳}	Test	cond	00	1	1000	1	Rn	0000	shifter operand
12	LDR	Load Register	cond	01	0	0100	1	Rn	Rd	offset_12
13	STR	Store Register	cond	01	0	0100	0	Rn	Rd	offset_12
14	B	Branch	cond	10	1	0	signed_immed_24			

جدول ۲- لیست دستورهای پردازنده

همانگونه که در جدول ۲ ملاحظه می شود هر دستور ISA پردازنده ARM دارای بخش های مختلفی است این بخش ها شامل موارد زیر است:

- ^{۱۱} نکته: در پردازنده ARM دستور NOP پیاده سازی نمی شود و در صورت نیاز از دستورات دیگر مانند AND یک عدد با خود آن استفاده می شود.
- ^{۱۲} در پردازنده ARM مقدار فوری همواره به صورت بدون علامت در نظر گرفته می شود. برای مقداردهی اعداد منفی از MVN استفاده می شود. این دستور مقدار shifter operand را مکمل ۱ می گیرد و در رجیستر مقصد ذخیره می کند.
- ^{۱۳} دستور TST برای مقایسه مقادیر منطقی مورد استفاده قرار می گیرد. این دستور عملیات AND را اجرا می کند و رجیستر وضعیت را به روزرسانی می کند. دستور TST رجیستر مقصد ندارد.



Mode: دسته دستور را تعیین می کند. تمامی دستورات محاسباتی در دسته 00 قرار می گیرند. دستورات حافظه در دسته ی 01 و دستورات پرش در دسته 10 قرار دارند. در این پردازنده ها دستورات ارتباط با پردازنده ی کمکی^{۱۴} نیز در نظر گرفته شده است که Mode آن برابر 11 است. OP-Code: کد دستورالعمل برای تعیین نوع دستور است. Mode به همراه OP-Code برای تشخیص دستورات در نظر گرفته می شود. I: نشاندهنده فوری بودن عملوند دوم است، در صورت یک بودن داده دوم فوری در نظر گرفته می شود. S: در صورت یک بودن S دستورات محاسباتی پس از اجرا ثبات وضعیت (state register) به روز می کنند. Cond: در پردازنده های ARM تمامی دستورات به صورت شرطی اجرا می شوند. در جدول ۳ لیست حالت های اجرای دستورات ذکر شده است. در صورتی که یک دستور به صورت غیرشرطی اجرا شود مقدار بیت های شرط برابر 1110 خواهد بود. در صورتی که شرط برقرار نباشد دستور همانند NOP هیچ کاری انجام نخواهد داد. مقدار 1111 نیز در نسل های مختلف پردازنده های ARM به صورت متفاوتی اجرا می شود که در پردازنده مورد نظر در آزمایشگاه نیازی به پیاده سازی آن نیست.

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See Condition code 0b1111	-

جدول ۳- کد شرط دستورات

Rd: نشاندهنده آدرس ثبات مقصد است. این آدرس در دستور STR به عنوان یکی از مقداری که باید در حافظه ذخیره شود مورد استفاده قرار می گیرد.

^{۱۴} Co-Processor

Rn: همواره به عنوان یکی از عملوندهای دستورات مورد استفاده قرار می گیرد.
shifter operand: برای عملوند شیفت در پردازنده ARM به سه شکل زیر پیاده سازی شده است:
-۱ ۳۲ بیت عدد فوری (32-bit immediate):

در این حالت مقدار بیت I برابر یک است. عدد ۸ بیتی imm8 در یک ظرف ۳۲ بیت قرار می گیرد سپس به اندازه دو برابر rotate_imm به راست چرخانده می شود (شکل ۱).

31	28	27	26	25	24	21	20	19	16	15	12	11	8	7	0
cond	0	0	1	opcode	S	Rn	Rd	rotate_imm	imm8						

شکل ۱: دستورالعمل از نوع ۳۲ بیت عدد فوری

Rotation	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x0																										7	6	5	4	3	2	1	0
0x1	1	0																															
0x2	3	2	1	0																													
0x3	5	4	3	2	1	0																											
0x4	7	6	5	4	3	2	1	0																									
0x5																																	
0x6																																	
0x7																																	
0x8																																	
0x9																																	
0xA																																	
0xB																																	
0xC																																	
0xD																																	
0xE																																	
0xF																																	

شکل ۲: نحوه چرخش عدد فوری



۲- شیفت فوری (Immediate shifts):

در این حالت بیت I و بیت چهارم دستورالعمل نیز صفر برابر صفر است. عملوند دوم از رجیستر خوانده می شود. سپس عدد خوانده شده براساس حالت شیفت (shift) به مقدار shift_imm شیفت داده می شود (شکل ۳). حالت های شیفت در جدول زیر قرار دارد.

وضعیت شیفت	توضیحات	مقدار
LSL	Logical shift left	00
LSR	Logical shift right	01
ASR	Arithmetic shift right	10
ROR	Rotate right	11

جدول ۴- وضعیت شیفت در دستورات شیفت فوری

31	28	27	26	25	24	21	20	19	16	15	12	11	7	6	5	4	3	0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	shift	0	Rm							

شکل ۳: دستورالعمل از نوع شیفت فوری

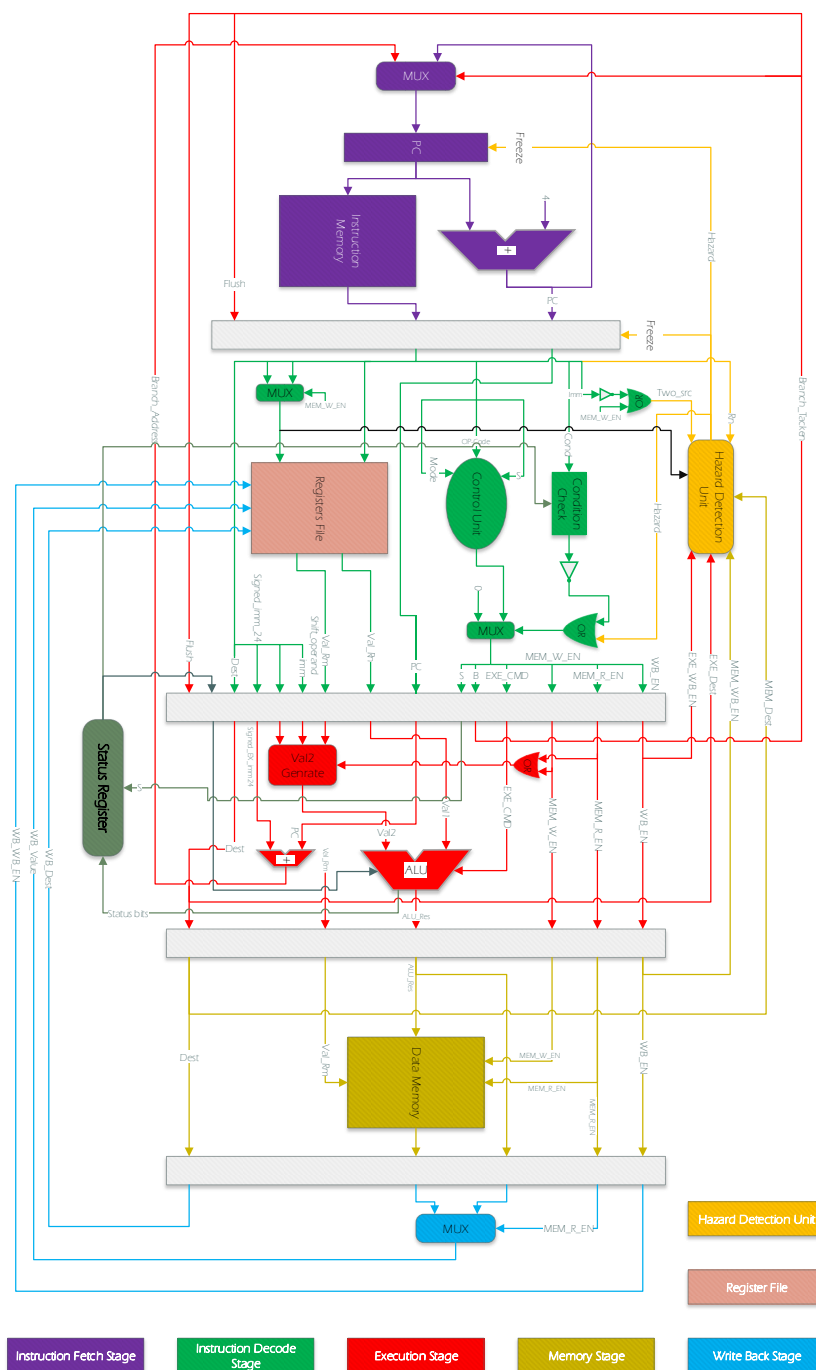
۳- شیفت ثباتی (Register shifts):

در این حالت بیت I برابر صفر است و عملوند دوم از رجیستر خوانده می شود. پس از آن عدد خوانده شده براساس حالت شیفت (shift) به مقدار رجیستر Rs شیفت داده می شود (شکل ۴). در پردازنده مورد استفاده در آزمایشگاه نیازی به پیاده سازی این حالت نیست.

31	28	27	26	25	24	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm							

شکل ۴: دستورالعمل از شیفت ثباتی

معماری پردازنده



شکل ۵- معماری کلی پردازنده ARM ساده شده



ثبات‌های عمومی^{۱۵}

- Register File در پردازنده ARM شامل ۱۶ ثبات ۳۲ بیت می شود که کاربردهای زیر را دارند:
- ثبات ۰ تا ۱۲ ثبات‌های عمومی پردازنده می‌باشند که در همه کاربردها استفاده می‌شوند.
 - ثبات ۱۳ به عنوان اشاره گر پشته^{۱۶} مورد استفاده قرار می گیرند. دستورات پشته مانند push و pop از این ثبات استفاده می کنند.
 - ثبات ۱۴ به عنوان آدرس بازگشت پس از دستور BL استفاده می‌شود. دستور BL یا Branch and Link معادل دستور Call در پردازنده‌های دیگر است.
 - ثبات ۱۵ به عنوان شمارنده برنامه مورد استفاده قرار می گیرد. در معماری ارائه شده در آزمایشگاه برای سادگی این رجیستر به مرحله واکنشی دستور^{۱۷} منتقل شده است.

ثبات وضعیت^{۱۸}

در پردازنده ARM یک ثبات برای نگهداری وضعیت کلی پردازنده در نظر گرفته شده است. این رجیستر Mode اجرای پردازنده و وضعیت اجرای دستورات در پردازنده را بیان می‌کند^{۱۹}. بیت‌های N (منفی بودن)، Z (صفر بودن)، C (رقم نقلی) و V (سرریز^{۲۰}) برای بررسی شرط مورد استفاده قرار می گیرد. در پیاده سازی پردازنده مورد نظر آزمایشگاه معماری کامپیوتر فقط بیت‌های Z، N و C و V پیاده‌سازی می‌شود. نوشتن در ثبات وضعیت با لبه پایین رونده انجام می‌شود.

31	30	29	28	27	26	25	24	23	20	19	16	15	10	9	8	7	6	5	4	0
N	Z	C	V	Q	Res	J	RESERVED	GE[3:0]	RESERVED	E	A	I	F	T	M[4:0]					

شکل ۶- ثبات وضعیت

^{۱۵} Register file

^{۱۶} Stack pointer

^{۱۷} Instruction Fetch (IF)

^{۱۸} Status Register

^{۱۹} برای مطالعه جزئیات این رجیستر به بخش A2.5 از ARM Architecture Reference Manual مراجعه کنید.

^{۲۰} Overflow or Underflow



خط لوله پردازنده

• مرحله واکنشی

در مرحله واکنشی دستورالعمل به یک ثابت برای نگه داری شماره برنامه (PC) نیاز است. همانطور که در شکل ۱ دیده می شود، این ثابت با توجه به نوع دستور، با $PC+4$ یا آدرس پرش (Branch Address) جایگزین می شود. همچنین از یک حافظه دستورالعمل (Instruction Memory) برای نگه داری دستورالعمل ها استفاده می شود. در شکل زیر نمونه ای از ورودی و خروجی های مرحله واکنشی و رجیستر پس از آن نشان داده شده است.

```
1 module IF_Stage (  
2     input clk, rst, freeze, Branch_taken ,  
3     input [31:0] BranchAddr,  
4     output [31:0] PC, Instruction  
5 );  
6  
7
```

```
1 module IF_Stage_Reg (  
2     input clk, rst, freeze, flush,  
3     input [31:0] PC_in, Instruction_in,  
4     output reg [31:0] PC, Instruction  
5 );  
6
```

• مرحله کد گشایی

در مرحله کدگشایی می بایست دستور به صورت کامل دیکد گردد، سیگنال های کنترلی ایجاد و مقادیر رجیستر خوانده شود. برای پیاده سازی مرحله کد گشایی انجام مراحل زیر الزامیست

۱- ایجاد مجموعه ثابت های عمومی

یک آرایه ۱۵ تایی با ثابت های ۳۲ بیتی، که دارای یک پورت نوشتن همگام با لبه پایین رونده و دو پورت خواندن ناهمگام است. نکته: در این پردازنده ثابت شماره صفر همواره مقدار ۰ را در خود نگهداری می کند. لیست پورتهای مجموعه ثابت ها در زیر نشان داده شده است.

```
1  
2 module RegisterFile (  
3     input clk, rst,  
4     input [3:0] src1, src2, Dest_wb,  
5     input [31:0] Result_WB,  
6     input writeBackEn,  
7     output [31:0] reg1, reg2  
8 );
```

۲- تکمیل مرحله کدگشایی

در این مرحله دستور به صورت کامل کدگشایی می گردد به گونه ای که دیگر در هیچ مرحله ای به Op-code نیازی نخواهد بود. از قسمت های اصلی این بخش پیاده سازی Control Unit به منظور ایجاد تمامی سیگنال های کنترلی پردازنده است. در مرحله کد گشایی همچنین کارهایی مانند تعیین سیگنال پرش، تعیین ورودی اول و دوم ALU، خواندن از رجیستر یا ارسال داده Immediate و تعیین آدرس رجیستر مقصد می بایست انجام گردد. در شکل زیر نمونه ای از ماژول کدگشایی و رجیستر بعد از آن نشان داده شده است.



```
1 module ID_Stage (  
2     input clk,rst,  
3     //from IF Reg  
4     input[31:0] Instruction,  
5     //from WB stage  
6     input[31:0] Result_WB,  
7     input writeBackEn,  
8     input[3:0] Dest_wb,  
9     //from hazard detect module  
10    input hazard,  
11    //from Status Register  
12    input[3:0] SR,  
13    //to next stage  
14    output WB_EN, MEM_R_EN, MEM_W_EN,B,S,  
15    output[3:0] EXE_CMD,  
16    output[31:0] Val_Rn, Val_Rm,  
17    output imm,  
18    output[11:0] Shift_operand,  
19    output[23:0] Signed_imm_24,  
20    output[3:0] Dest,  
21    //to hazard detect module  
22    output[3:0] src1,src2,  
23    output Two_src  
24 );  
25
```

```
1 module ID_Stage_Reg (  
2     input clk,rst,flush,  
3     input WB_EN_IN, MEM_R_EN_IN, MEM_W_EN_IN,  
4     input B_IN,S_IN,  
5     input[3:0] EXE_CMD_IN,  
6     input[31:0] PC_IN,  
7     input[31:0] Val_Rn_IN, Val_Rm_IN,  
8     input imm_IN,  
9     input[11:0] Shift_operand_IN,  
10    input[23:0] Signed_imm_24_IN,  
11    input[3:0] Dest_IN,  
12  
13    output reg WB_EN, MEM_R_EN, MEM_W_EN,B,S,  
14    output reg[3:0] EXE_CMD,  
15    output reg[31:0] PC,  
16    output reg[31:0] Val_Rn, Val_Rm,  
17    output reg imm,  
18    output reg[11:0] Shift_operand,  
19    output reg[23:0] Signed_imm_24,  
20    output reg[3:0] Dest  
21 );
```

• مرحله اجرا

اجرای تمامی دستورات حسابی منطقی در این مرحله انجام می شود. همچنین آدرس درس پرش و آدرس ذخیره یا خواندن دستورات از حافظه داده در این مرحله انجام می شود.

```
1 module EXE_Stage(  
2     input clk,  
3     input[3:0] EXE_CMD,  
4     input MEM_R_EN, MEM_W_EN,  
5     input [31:0] PC,  
6     input[31:0] Val_Rn, Val_Rm,  
7     input imm,  
8     input[11:0] Shift_operand,  
9     input[23:0] Signed_imm_24,  
10    input [3:0] SR,  
11  
12    output[31:0] ALU_result,Br_addr,  
13    output[3:0] status  
14 );
```

```
1 module EXE_reg(  
2     input clk, rst, WB_en_in, MEM_R_EN_in, MEM_W_EN_in,  
3     input[31:0] ALU_result_in, ST_val_in,  
4     input[3:0] Dest_in,  
5     output reg WB_en, MEM_R_EN, MEM_W_EN,  
6     output reg[31:0] ALU_result, ST_val,  
7     output reg[3:0] Dest  
8 );
```

در پردازنده های مختلف مرحله اجرا شامل واحدهایی همچون واحد حساب و منطق (ALU)، FMA، X87، Cryptography module و... است. در پردازنده مورد نظر در این آزمایش مرحله اجرا شامل ALU و محاسبه آدرس دستور پرش خواهد بود. ALU دارای دو ورودی داده، یک خروجی داده و یک ورودی چهار بیتی است که توسط Control Unit تولید شده و تعیین کننده عملیات ALU است. این ورودی کنترلی در جدول ۵ مشخص شده است.



Instruction	ALU Command	Operation
MOV	0001	result = in2
MVN	1001	result = ~in2
ADD	0010	result = in1 + in2
ADC	0011	result = in1 + in2 + C
SUB	0100	result = in1 - in2
SBC	0101	result = in1 - in2 - ~C
AND	0110	result = in1 & in2
ORR	0111	result = in1 in2
EOR	1000	result = in1 ^ in2
CMP	0100	result = in1 - in2
TST	0110	result = in1 & in2
LDR	0010	result = in1 + in2
STR	0010	result = in1 + in2
B	XXXX	

جدول ۵- ریز دستورهای واحد حساب و منطق

• مرحله حافظه

در مرحله حافظه داده‌ها از یک حافظه RAM شبیه سازی شده با سیگنال‌های MEM_R_EN و MEM_W_EN به ترتیب خوانده و در آن نوشته می‌شود. این سیگنال‌ها در مرحله گذرایی توسط Control unit تولید و همراه با دستور در پایپ به جلو حرکت ارسال می‌شود. حافظه داده از آدرس ۱۰۲۴ شروع می‌شود و آدرس دهی براساس بایت خواهد بود. در هر مرحله خواندن از حافظه ۳۲ بیت داده خوانده یا نوشته می‌شود و دسترسی به تک بایت امکانپذیر نیست.

❖ خواندن و نوشتن فقط از آدرس‌های مضرب ۴ (به دلیل ۳۲ بیتی بودن معماری) انجام می‌شود. به طور مثال: در ازای خواندن از آدرس‌های ۱۰۲۴، ۱۰۲۵، ۱۰۲۶ و ۱۰۲۷ نتایج یکسانی خوانده می‌شود یعنی ۴ بایت از آدرس ۱۰۲۴.

حجم حافظه را ۲۵۶ بایت در نظر بگیرید.

```
1 module Memory(  
2     input clk, MEMread, MEMwrite,  
3     input[31:0] address, data,  
4     output[31:0] MEM_result  
5 );  
  
1 module MEM_reg(  
2     input clk, rst, WB_en_in, MEM_R_en_in,  
3     input[31:0] ALU_result_in, Mem_read_value_in,  
4     input[3:0] Dest_in,  
5     output reg WB_en, MEM_R_en,  
6     output reg[31:0] ALU_result, Mem_read_value,  
7     output reg[3:0] Dest  
8 );
```

• مرحله باز نویسی

- در این مرحله با سیگنال WB_EN داده ارسالی از مرحله حافظه یا اجرا در ثبات مقصد از ثبات‌های عمومی نوشته خواهد. سیگنال WB_EN توسط واحد کنترل همراه با دستور به جلو ارسال می‌گردد. همچنین به کمک سیگنال MEM_R_EN نیز نوع



دستور (حافظه‌ای یا محاسباتی) تشخیص داده می‌شود و مقدار خوانده شده از حافظه یا مقدار محاسبه شده از ALU در ثبات مقصد نوشته می‌شود.

```
1 module WB_stage(  
2     input [31:0] ALU_result, MEM_result,  
3     input MEM_R_en,  
4     output [31:0] out  
5 );
```

اجرای برنامه محک

برای تست پردازنده باید برنامه محک در Instruction Memory قرار گیرد و نتایج اجرا به همراه تعداد سیکل‌های اجرا ثبت شود. دستورات برنامه محک به صورت دودویی و اسمبلی در پیوست ۱ قرار داده شده است. در بخش اول این برنامه تمامی دستورات پیاده سازی شده را تست می‌کند و پس از آن یک الگوریتم مرتب‌سازی حبابی^{۲۱} پیاده شده است. سپس داده‌های مرتب شده را در رجیسترهای R1 تا R4 بارگذاری می‌کند. در صورت صعودی بودن رجیسترهای R1 تا R4 برنامه به درستی اجرا شده است.

استاندارد دستورات اسمبلی ARM به صورت زیر بیان می‌شوند:

Instruction<Cond><S> R_d, R_n, <Shifter Operand>

مثال:

ADD R0, R1, #10

مجموع R1 و ۱۰ را در R0 ذخیره می‌کند. این دستور از نوع ۳۲ بیت عدد فوری است.

ADDEQ R0, R1, #10

این دستور معادل دستور قبل است با این تفاوت که در صورتی که شرط EQ (جدول ۳) برقرار باشد اجرا می‌شود.

ADDS R0, R1, #10

این دستور علاوه بر ذخیره‌ی مقدار مجموع R1 و ۱۰ را در R0، ثبات وضعیت را بروز رسانی می‌کند.

ADDEQS R0, R1, R2

این دستور به صورت شرطی و با بروز رسانی ثبات وضعیت مجموع R2 و R1 را محاسبه می‌کند و در R0 ذخیره می‌کند. این دستور از نوع شیفت فوری است و مقدار شیفت آن صفر خواهد بود.

ADD R0, R1, R2, LSL #2

^{۲۱} Bobble Sort



این دستور ابتدا R2 را دو بیت به چپ شیفت فوری می‌دهد (جدول ۴) سپس حاصل را با R1 جمع می‌کند.

گزارش کار

- در ابتدای گزارش کار باید مدار طراحی شده در سطح عملکردی توضیح داده شود، سپس معماری آن در سطح RTL را با توضیحات کامل نوشته شود.
- در قسمت بعد کد Verilog معادل با RTL طراحی شده توضیح داده شود و نتایج شبیه سازی برای نشان دادن درستی کد آورده شود (به ازای هر دستور یک نتیجه به همراه تصویری از SignalTap II ارائه شود).
- پس از آن نتایج سنتز آورده شود و مدار RTL استخراج شده از Quartus II با مدار RTL طراحی شده در قسمت اول مقایسه شود و تفاوت ها را توضیح دهید.
- نتایج برنامه ریزی روی برد را توضیح دهید.
- تصویر گزارش کامپایل (Compilation Report)
- جدولی حاوی موارد زیر را گزارش نمایید:
 - تعداد کل المان‌های منطقی استفاده شده در پروژه (Total Logic Elements)
 - تعداد المان‌های منطقی استفاده شده در مدارات ترتیبی (Total Combinational functions)
 - تعداد المان‌های منطقی استفاده شده توسط رجیسترها (Dedicated Logic registers)
 - زمان اجرای برنامه: زمان اجرای برنامه برابر با تعداد کلاک‌هایی است که PC برای اولین بار به دستور "1-JMP" می‌رسد.
 - میزان CPI (تعداد کلاک‌های اجرای برنامه بر دستور العمل).
- در قسمت آخر گزارش کار باید مشکلاتی که هنگام کدنویسی داشته‌اید، همچنین خطاهای زمان کامپایل و سنتز نوشته شود و راهکارهایی که این مشکلات و خطاها را برطرف نموده‌اید را بیان کنید.



پیوست : برنامه محک

کد ماشین به همراه اسمبلی و نتایج:

1. 32b1110_00_1_1101_0_0000_0000_000000010100; //MOV	R0 ,#20	//R0 = 20
2. 32b1110_00_1_1101_0_0000_0001_101000000001; //MOV	R1 ,#4096	//R1 = 4096
3. 32b1110_00_1_1101_0_0000_0010_000100000011; //MOV	R2 ,#0xC0000000	//R2 = -1073741824
4. 32b1110_00_0_0100_1_0010_0011_000000000010; //ADDS	R3 ,R2,R2	//R3 = -2147483648
5. 32b1110_00_0_0101_0_0000_0100_000000000000; //ADC	R4 ,R0,R0	//R4 = 41
6. 32b1110_00_0_0010_0_0100_0101_000100000100; //SUB	R5 ,R4,R4,LSL #2	//R5 = -123
7. 32b1110_00_0_0110_0_0000_0110_000010100000; //SBC	R6 ,R0,R0,LSR #1	//R6 = 10
8. 32b1110_00_0_1100_0_0101_0111_000101000010; //ORR	R7 ,R5,R2,ASR #2	//R7 = -123
9. 32b1110_00_0_0000_0_0111_1000_000000000011; //AND	R8 ,R7,R3	//R8 = -2147483648
10. 32b1110_00_0_1111_0_0000_1001_000000000110; //MVN	R9 ,R6	//R9 = -11
11. 32b1110_00_0_0001_0_0100_1010_000000000101; //EOR	R10,R4,R5	//R10 = -84
12. 32b1110_00_0_1010_1_1000_0000_000000000110; //CMP	R8 ,R6	
13. 32b0001_00_0_0100_0_0001_0001_000000000001; //ADDNE	R1 ,R1,R1	//R1 = 8192
14. 32b1110_00_0_1000_1_1001_0000_000000001000; //TST	R9 ,R8	
15. 32b0000_00_0_0100_0_0010_0010_000000000010; //ADDEQ	R2 ,R2,R2	//R2 = -1073741824
16. 32b1110_00_1_1101_0_0000_0000_101100000001; //MOV	R0 ,#1024	//R0 = 1024
17. 32b1110_01_0_0100_0_0000_0001_000000000000; //STR	R1 ,[R0],#0	//MEM[1024] = 8192
18. 32b1110_01_0_0100_1_0000_1011_000000000000; //LDR	R11,[R0],#0	//R11 = 8192
19. 32b1110_01_0_0100_0_0000_0010_000000000100; //STR	R2 ,[R0],#4	//MEM[1028] = -1073741824
20. 32b1110_01_0_0100_0_0000_0011_000000001000; //STR	R3 ,[R0],#8	//MEM[1032] = -2147483648
21. 32b1110_01_0_0100_0_0000_0100_000000001101; //STR	R4 ,[R0],#13	//MEM[1036] = 41
22. 32b1110_01_0_0100_0_0000_0101_000000010000; //STR	R5 ,[R0],#16	//MEM[1040] = -123
23. 32b1110_01_0_0100_0_0000_0110_000000010100; //STR	R6 ,[R0],#20	//MEM[1044] = 10
24. 32b1110_01_0_0100_1_0000_1010_000000000100; //LDR	R10,[R0],#4	//R10 = -1073741824
25. 32b1110_01_0_0100_0_0000_0111_000000011000; //STR	R7 ,[R0],#24	//MEM[1048] = -123
26. 32b1110_00_1_1101_0_0000_0001_000000000100; //MOV	R1 ,#4	//R1 = 4
27. 32b1110_00_1_1101_0_0000_0010_000000000000; //MOV	R2 ,#0	//R2 = 0
28. 32b1110_00_1_1101_0_0000_0011_000000000000; //MOV	R3 ,#0	//R3 = 0
29. 32b1110_00_0_0100_0_0000_0100_000100000011; //ADD	R4 ,R0,R3,LSL #2	
30. 32b1110_01_0_0100_1_0100_0101_000000000000; //LDR	R5 ,[R4],#0	
31. 32b1110_01_0_0100_1_0100_0110_000000000100; //LDR	R6 ,[R4],#4	
32. 32b1110_00_0_1010_1_0101_0000_000000000110; //CMP	R5 ,R6	
33. 32b1100_01_0_0100_0_0100_0110_000000000000; //STRGT	R6 ,[R4],#0	
34. 32b1100_01_0_0100_0_0100_0101_000000000100; //STRGT	R5 ,[R4],#4	
35. 32b1110_00_1_0100_0_0011_0011_000000000001; //ADD	R3 ,R3,#1	
36. 32b1110_00_1_1010_1_0011_0000_000000000011; //CMP	R3 ,#3	
37. 32b1011_10_1_0_11111111111111111111111111111111 ; //BLT	#-9	
38. 32b1110_00_1_0100_0_0010_0010_000000000001; //ADD	R2 ,R2,#1	



دستور کار آزمایشگاه معماری کامپیوتر
بخش سخت افزار، دانشکده برق و کامپیوتر، دانشگاه تهران
آزمایش اول: پیاده سازی پردازنده ARM



39.	32b1110_00_0_1010_1_0010_0000_000000000001; //CMP	R2,R1	
40.	32b1011_10_1_0_111111111111111110011 ; //BLT	#-13	
41.	32b1110_01_0_0100_1_0000_0001_000000000000; //LDR	R1,[R0],#0	//R1 = -2147483648
42.	32b1110_01_0_0100_1_0000_0010_000000000100; //LDR	R2,[R0],#4	//R2 = -1073741824
43.	32b1110_01_0_0100_1_0000_0011_000000001000; //LDR	R3,[R0],#8	//R3 = 41
44.	32b1110_01_0_0100_1_0000_0100_000000001100; //LDR	R4,[R0],#12	//R4 = 8192
45.	32b1110_01_0_0100_1_0000_0101_000000010000; //LDR	R5,[R0],#16	//R5 = -123
46.	32b1110_01_0_0100_1_0000_0110_000000010100; //LDR	R6,[R0],#20	//R4 = 10
47.	32b1110_10_1_0_111111111111111111111111 ; //B	#-1	