دانشگاه خلیج فارس
دانشکده مهندسی سیستم‌های هوشمند و علوم داده

پایان‌نامه‌ی کارشناسی
مهندسی کامپیوتر

عنوان:

ماشین مجازی زاگرس

نگارش:

آرین دشتی

استاد راهنما:

دکتر ابراهیم صحافی‌زاده

شهریور ۱۴۰۱


بسم الله الرحمن الرحیم

چکیده

زاگرس یک سامانه محاسبات مجازی قابل جاسازی[۱] است. به اندازه‌ای سبک[۲] است که در سیستم‌های نهفته استفاده شود و به اندازه‌ای سریع است که مورد استفاده برنامه‌های سروری قرار گیرد. منابع سخت‌افزاری که انتزاع می کند همگی قابل تنظیم است بنابر این می تواند برای نیاز های متفاوت پیکربندی شود. از SWIG[۳] برای ایجاد اتصالات برای اکثر زبان‌های برنامه نویسی مطرح استفاده می کند و می‌توان ازین طریق حتی یک پوسته‌ی واکنشی[۴]، قابلیت بروزرسانی زنده[۵]، و همینطور قابلیت برنامه‌نویسی و بررسی بلادرنگ[۶] نیز به آن اضافه کرد.

کلیدواژه‌ها: ماشین مجازی، سیستم های نهفته، بروزرسانی زنده

---

[1] Embeddable
[2] Lightweight
[3] Simplified Wrapper and Interface Generator
[4] Reactive Shell
[5] Live Upgrade
[6] Real-time

فهرست مطالب

فهرست شکل‌ها

فهرست جدول‌ها

فصل ۱

بررسی اجمالی

۱-۱ معرفی

نفوذ کامپیوتر در زندگی روی زمین روز به روز بیشتر می شود. با بیشتر شدن این نفوذ و کامپیوتری شدن عناصر زندگی، نیاز به متخصص‌ها هم بیشتر می شود. هرچند عرضه‌ی متخصصین برنامه‌نویسی کامپیوتر با پیشرفت تکنولوژی بیشتر می شود، اما عرضه‌ی متخصصین دامنه‌ی عناصری که به تکنولوژی روی می آورند ثابت می‌ماند. به همین دلیل است که طراحی دامنه محور [۱] امروزه مهارت مهمی برای برنامه‌نویسان بکند شده است، تا بتوانند بیشترین استفاده و کوتاه ترین حلقه بازخورد[۲] را از متخصصین دامنه داشته باشند. بعضی دامنه‌ها پویا[۳] اند و به دلیل نیازمندی های روز افزون کارایی و امنیتی باید به آسانی تغییر، ترکیب و گسترش داده شوند. برای این دامنه ها زبان‌های خاص دامنه[۴] طراحی می شوند. طراحی زبان‌های خاص دامنه، مانند طراحی هر زبان برنامه نویسی دیگری سختی هایی را بدنبال دارد. دانشمندان لایه هایی انتزاع[۵] مستقل تعریف کرده اند[۶] تا با حل کردن جدا از هم این مسائل از سختی کار بکاهند. زاگرس یک ماشین مجازی است که در اولین لایه ی عقب بندی[۷] نشسته و کد میانی[۸] به

[1] Domain Driven Design
[2] Feedback Loop
[3] Dynamic
[4] Domain Specific Language
[5] Abstraction Layer
[۶] همان ۷ لایه توسعه کامپایلر
[7] Backend
[8] Intermediate Representation

صورت بایت‌کد[9] ارائه می دهد، برنامه ای که از مرحله ی تحلیل معنایی[10] یک کامپایلر به درستی گذشته را دریافت، ورودی/خروجی ها را سر هم و برنامه را تفسیر می کند.

۱-۲ طراحی دامنه محور

طراحی دامنه محور رویکردی برای توسعه نرم افزار است. این رویکرد توسعه را بر برنامه نویسی یک مدل دامنه متمرکز می کند، مدلی که درکی غنی از فرآیندها و قوانین آن دامنه را دارد. این نام برگرفته از کتابی از اریک ایوانز[11] در سال ۲۰۰۳ است که این رویکرد را از طریق فهرستی از الگوها توصیف می کند. از آن زمان جامعه‌ای از متخصصان این ایده‌ها را بیشتر توسعه داده‌اند، کتاب‌های مختلف دیگری نوشتند و دوره‌های آموزشی را طراحی کرده اند. این رویکرد مخصوصا برای حوزه های پیچیده مناسب است، جایی که بسیاری از منطق های اغلب درهم و برهم نیاز به سازماندهی دارند.

۱-۲-۱ زبان‌های خاص دامنه

یک زبان خاص دامنه، یک زبان کامپیوتری است که به جای یک زبان همه منظوره که هر نوع مسأله نرم افزاری را هدف قرار می دهد، برای نوع خاصی از مشکل هدف قرار می گیرد. تقریباً از روزی که محاسبات انجام داده می‌شده، درباره زبان‌های خاص دامنه هم صحبت می‌شده.

زبان‌های خاص دامنه در دنیای کامپیوتر بسیار متداول هستند: مثال هایی شامل CSS، عبارات منظم، make، ant، SQL بخش های زیادی از Rails، انتظارات در JMock، زبان dot از graphviz، فایل پیکربندی strut و غیره...

یک تمایز مهم و مفید بین زبان‌های خاص دامنه داخلی و خارجی بودن است. زبان‌های خاص دامنه داخلی روش های خاصی برای استفاده از زبان میزبان هستند تا به زبان میزبان حس یک زبان خاص را بدهد. این رویکرد مدتهاست که بخشی از سنت Lisp بوده است و در دهه گذشته توسط جامعه Ruby نویسان تولدی دوباره یافته. اگرچه معمولاً زبان‌های خاص داخلی با میزبان‌های کم تشریفات شکل ساده‌تری دارند، اما می‌توانید زبان‌های خاص داخلی مؤثری را در زبان‌های رایج‌تر مانند جاوا و

[9] Bytecode
[10] Semantic Analysis
[11] Eric Evans

سی‌شارپ نیز توسعه داد. به زبان‌های خاص داخلی عناوینی چون زبان‌خاص جای‌گذاری شده[۱۲] یا رابط های روان[۱۳] نیز می دهند.

زبان‌های خاص های خارجی سینتکس مخصوص خود را دارند و یک تجزیه‌کننده[۱۴] کامل برای پردازش آنها نوشته می شود. این اتفاق را به کرار در یونیکس[۱۵] می بینیم. یک راه در رو برای این مسآله استفاده از ساختار های XML و YAML و پردازش آن ها به عنوان درخت نحو انتزاعی[۱۶] تا خود مجبور به نوشتن تجزیه‌کننده نباشیم.

امروزه رایج ترین زبان‌های خاص دامنه متنی هستند، اما زبان‌های خاص می توانند گرافیکی نیز باشند. زبان‌های خاص گرافیکی به میزکار زبان[۱۷] نیازمندند. میزهای کار زبان به آهستگی، اما پیوسته در حال محبوب تر شدن هستند. این ابزارها پتانسیل را دارند که رویکرد برنامه نویسی ما را به شدت بهبود بخشند و حتی شرکت JetBrains نیز با ارائه MPS JetBrains یک میزکار زبان ارائه می دهد.

زبان‌های خاص دامنه را می توان با تفسیر یا تولید کد پیاده سازی کرد. تفسیر (خواندن زبان به صورت یک اسکریپت و اجرای آن در زمان اجرا) معمولاً ساده‌ترین رویکرد است است، اما تولید کد گاهی ضروری است.

۱-۳ تفسیر زبان‌های برنامه نویسی

در طیف تفسیر زبان‌های برنامه نویسی دو انتها وجود دارد. یکی تفسیر مستقیم درخت تجزیه انتزاعی[۱۸] و دیگری کامپایلر های پیش‌تر از زمان[۱۹] هستند. اولی سریع ترین روش در پیاده سازی و کند ترین روش در اجرای برنامه است و آخری دقیقا برعکس. مانند هر تصمیم دیگری در مهندسی، با سبک و سنگین کردن باید به میانه ای مناسب برای نیازمندی های خود برسیم.

[12] Embedded DSL
[13] Fluent Interfaces
[14] Parser
[15] Unix
[16] Abstract Syntax Tree
[17] Language Workbench
[18] Abstract Syntax Tree
[19] Ahead of time

۱-۳-۱ تفسیر مستقیم درخت تجزیه انتزاعی

پیمایش درخت تجزیه انتزاعی و تفسیر آن اولین راه حلی است که یک مهندس در کار یا دانشگاه می آموزد و دلایل خوبی برای آن وجود دارد. سرعت پیاده‌سازی این تکنیک بسیار بالاست و پیچیدگی آن بسیار کم است. به همین خاطر پیاده‌سازی کننده می تواند وقت بیشتری را به تحقیق و آزمایش معنا[۲۰] زبانی که پیاده سازی می کند اختصاص دهد و همینطور با تست کردن بیشتر از صحت مفسر خود اطمینان یابد. از آنجایی که این مفسر ها زبان مهمان خود را کامپایل نمی کنند و فقط خودشان کامپایل می شوند چرخه ی بازخورد توسعه زبان را بسیار کوتاه‌تر می کنند و مناسب بوت‌استرپینگ[۲۱] کردن کامپایلر ها هستند. همینطور این کوتاه بودن زمان بارگذاری آن‌ها را مناسب برای بکندهای وب بدون‌سرور[۲۲] می کند. در قبال این خوبی ها، چنین مفسرهایی دو ضعف دارند. اولین ضعف آن ها سرعتشان در تفسیر است. درخت‌تجزیه انتزاعی از سخت افزار بسیار دور است پس نمی تواند از بهینه سازی های سخت افزار استفاده کند، مجبور است از حافظه ی هیپ[۲۳] استفاده کند پس پیمایش آن خطاهای زیادی برای حافظه میانی ایجاد می کند و این بیراهه روی در حافظه از سرعت می کاهد. دومین ضعفشان سختی در سریالیز شدن است. ساختهار های درختی از ارجاع[۲۴] های زیادی دارند و ذخیره سازی بهینه ی این ارجاع‌ها برای انتقالشان در شبکه باید یا از لحاظ زمانی گران تمام شود یا از لحاظ حجمی.

۱-۳-۲ کامپایل کردن

اگر ما یک فندک را برداریم و با ماشین زمان به ۱۰۰۰ سال پیش برویم، مردم عادی ما را به جادوگری متهم می کنند. حال اگر ما GNU Compiler Collection را برداشته و با همان ماشین ۱۰۰۰ سال به آینده برویم، باز هم به جادوگری متهم می شویم. کامپایلر ها یکی از باشکوه ترین و پیچیده ترین دستاورد های مهندسی کامپیوتر هستند. نوشتن یک کامپایلر پیش‌تر از زمان یک مناسک‌گذار[۲۵] برای مهندسین کامپیوتر است. به همین دلیل سرعت پیاده سازی آن بسیار پایین، حلقه بازخورد آن بسیار طولانی و تست کردن آن دشوار است. کامپایلر ها بهینه سازی بیشینه را هدف قرار می دهند به همین خاطر کامپایل کردن یک کار وقت گیر، و برنامه ی کامپایل شده یک فایل حجم‌گیر است. کامپایلر ها

[20] Semantics
[21] Bootstrapping
[22] Serverless
[23] Heap
[24] Reference
[25] Rite of Passage

سیستم عامل و معماری سخت افزار را به صورت مستقیم هدف قرار می دهند به همین خاطر معمولا Cross-platform[۲۶] نیستند. وقت گیری کامایل کردن را عملی نامناسب برای نمونه‌سازی می کند. حجم گیری برنامه ها را نا مناسب برای جابجایی در شبکه می کند. Cross-platform نبودن هم از قابلیت انتقال[۲۷] بکند وب ما می کاهد. انتقال کدی که به زبان ماشین نوشته شده نیز بسیار از لحاظ امنیتی ریسک پذیر است زیرا برنامه به تمامی اختیارات سطح‌کاربر دسترسی دارد. عیب‌یابی[۲۸] کد ماشین نیز بسیار سخت و در نوعی متفاوت از کامپایل کردن انجام می شود. برنامه ای که منتشر می شود دیگر اطلاعات لازم برای عیب‌یابی را در خود ندارد.

۱-۳-۳ کامپایلرهای مفسر

برعکس فمنیسم، کامپایلر ها و مفسر ها می توانند به یک میانه متعادل برسند. همانند سیستم عامل، در تفسیر برنامه ها نیز کلید مساله تعریف یک ماشین مجازی است. ماشین مجازی مانند یک پردازنده واقعی دنباله ای از بایت ها را دریافت می کند. به همین دلیل می تواند از بهینه سازی های مختص سخت افزار استفاده کند. این دنباله از بایت یک آرایه در حافظه است و آرایه مساعد ترین و مورد علاقه ترین ساختمان داده ی حافظه های میانی است. سریالیز کردن و دسریالیز کردن دنباله ای از بایت ها کار بسیار بهینه است. درست است که ترجمه کردن درخت تجزیه انتزاعی به دنباله ی بایت به یک ماشین مجازی کاری دشوار تر از اجرای مستقیم آن درخت است اما بسیار ساده تر از ترجمه درخت به زبان ماشین است. چون همیشه فرصت بیشتری برای بهینه سازی در مرحله ی تفسیر کردن باقی می ماند، نیازی نیست همه ی بهینه سازی ها را در مرحله ترجمه انجام دهیم پس حلقه ی بازخورد کوتاه خواهد بود. می توان این دنباله ها را فشرده کرد و یا حتی نوعی فشرده از آن را تعریف کرد. می توانیم فقط برای هر سیستم عامل و معماری فقط یک برنامه مفسر ماشین مجازی بنویسیم و آن مفسر همگی برنامه هایی که در آینده برای آن ماشین مجازی نوشته می شود را اجرا کند و اینگونه به Cross-platform بودن برسیم. می توانیم سطح های متفاوت از دسترسی برای عملیات‌های تعریف شده در دنباله بایت ها تعریف کنیم و به سطوح دسترسی ای مختص دامنه ی خود دست یابیم. همینطور ماشین مجازی برعکسم ماشین حقیقی می تواند در هر لحظه از زمان متوقف و بررسی شود، عیب یابی شده و وضعیت آن تغییر داده شود. وقتی که نیازمند به بهینگی بیشینه باشیم می توانیم بجای تفسیر کردن، دنباله بایت ماشین مجازی را در لحظه،

[۲۶] از چند سیستم عامل و معماری سخت افزاری پشتیبانی نمی کنند

[27] Portability
[28] Debug

با اطلاعاتی که از وضعیت حاضر برنامه داریم، به زبان ماشین کامپایل کنیم[۲۹] و به بهینگی حتی بیشتری از حالت کامپایل کردن پیش‌از زمان برسیم. هرچند این کار از سرعت بارگذاری کم کرده و برنامه را نامناسب برای بکند های بدون سرور می کند. کامپایل کردن در لحظه البته نیازمند به دسترسی نوشتن در حافظه اجرایی است که همگی سیستم‌ها یا سیستم عامل ها این اجازه را به ما نمی دهند. نگهداری اطلاعات بهینه سازی نیز مصرف حافظه را دو چندان می کند.

۱-۴ هدف

زاگرس قصد دارد یک ماشین مجازی سبک و قابل جاسازی[۳۰] برای زبان‌های خاص دامنه که قصد به استفاده از کامپایلرهای مفسر دارند ارائه دهد.

۱-۵ ساختار گذارش

در فصل ۲ پیش‌زمینه ای از نحوه‌ی پیاده‌سازی بخش‌به‌بخش ماشین‌های مجازی شرح داده شده. در فصل ۳ مواصفات پیکر زاگرس و روش عملکردش بیان شده. در فصل ۴ از نحوه پیاده‌سازی مرجع مطرح شده. فصل ۵ راجع به نتیجه گیری صحبت می کند.

[29] Just in time compilation
[30] Embedding

فصل ۲

پیش‌زمینه

۲-۱ جدال ثبات‌ها

در ماشین های حقیقی، ثبات ها ذخایری هستند که پردازنده سرعت دسترسی بسیار بالایی به آنها دارد. ساختن ثبات‌ها ها گران تمام می شوند به همین دلیل تعداد آن ها محدود است. پشته، ساختاری در حافظه است که در انجام عملیات ها به یاری پردازنده می آید. هر عملگری که در لحظه نیازی به آن نباشد می تواند در پشته ذخیره شود، تا هنگام نیاز از پشته فراخوانی شود. در ماشین های مجازی، ثبات ها مجازی هستند پس در ظاهر محدودیتی راجع به تعداد آن ها وجود ندارد. اما در واقع اینطور نیست. اول از همه تعداد ثبات‌ها قالب دنباله بایت‌های برنامه را تغییر می دهد. اندازه ی هر کلمه[۱] ی دنباله و همینطور نحوه‌ی واکشی[۲] ماشین مجازی به تعداد رجیستر ها و قالب بایت‌ها وابسته است. هر کامپایلری که بخواهد زبانی سطح بالاتر را به بایت‌های ماشین مجازی ترجمه کند نیز مجبور به استفاده از یک الگوریتم پیچیده ی تخصیص ثبات [۳] می شود. به علاوه اگر ماشین مجازی بخواهد از ترجمه‌ی سروقت [۴] استفاده کند، تفاوت تعداد رجیسترهای ماشین مجازی و سخت‌افزار مقصد عمل ترجمه را بسیار سخت می کند. به همین دلیل است که ماشین های مجازی یا اصلا رجیستر ندارند و از پشته استفاده می کنند، یا بی نهایت (یا به مقداری بزرگ به اندازه ای که قالب بایت‌ها اجازه دهد) رجیستر

[1] Word
[2] Fetch
[3] Register Allocation
[4] Just in time compilation

دارند. در ادامه به ذکر مثال و مقایسه این دو نوع ماشین می پردازیم.

۲-۱-۱ ماشین‌های ثباتی

دالویک[۵] (ماشین مجازی جاوا برای اندروید)، LuaJIT (یک کامپایلر سروقت برای لوآ)، BEAM (ماشین مجازی ارلنگ[۶]) مثالهایی از ماشین‌های ثباتی معروف هستند. ماشین‌های ثباتی مجبورند محل حضور عملوند های عملگر ها را در زبان بایتی خود رمزنگاری و رمزگشایی کنند. به همین دلیل حجم و پیچیدگی اعمال ها زیاد می شود و پیاده سازی آن‌ها سخت تر است. در عوض چون ماشین های ثباتی میتوانند بی ن هایت ثبات و در نتیجه بی‌نهایت عملوند داشته باشند، کامپایلر می تواند از قبل تعداد مورد نیاز ثبات خودش را درخواست کند و نیازی به چرخش و تکرار عملوندها ندارند. در نتیجه ماشین‌های ثباتی نیاز به انجام عملیات‌های درون ماشینی کمتری برای اجرای الگوریتم ها دارند. اگر کامپایل کردن سر وقت را در نظر نگیریم ماشین های ثباتی بهینگی بیشتری نسبت به ماشین های پشته ای دارند.

۲-۱-۲ ماشین‌های پشته ای

ماشین مجازی جاوا، CLR (ماشین مجازی سی‌شارپ)، WASM (وب اسمبلی) مثالهایی از ماشین های پشته‌ای معروف هستند. در ماشین های پشته ای عملوند های هر عملیات همواره در روی پشته قرار دارد، به همین دلیل نیازی به رمزنگاری و رمزگشایی محل حضور عمولند در زبان بایتی نیست و کد تولید شده برای این ماشین‌ها حجم کمتری می گیرد و پیاده سازی آن ها آسان تر است. در عوض، چون با انجام هر عملیات عملوند ها از روی پشته برداشته می شوند و عملیات ها همگی حالت پسوندی دارند نیاز به چرخش و تکرار عملوند ها زیاد است. در نتیجه ماشین های پشته ای نیاز به انجام عملیات‌های درون ماشینی بیشتری برای انجام الگوریتم ها دارند و به همین خاطر است که در بعضی ماشین های پشته ای، برا مقاصد خاص پشته های خاصی تعریف می شود، مثلا پشته‌ی آدرس توابع یا پشته‌ی عملیات های اعشاری. اگر کامپایل‌کردن سروقت را در نظر نگیریم ماشین های پشته ای ردپای حافظه‌ی کمتری نسبت به ماشین‌های ثباتی دارند.

[5] Dalvik
[6] Erlang

۲-۲ حافظه در ماشین‌های مجازی

بعضی ماشین های مجازی مثل WASM دید خطی نسبت به حافظه دارند. بعضی ماشین‌های مجازی مثل ماشین مجازی جاوا بخشی از حافظه را خطی (پشته) و بعضی را بخشی را غیر خطی می‌بینند (هیپ[۷]). در حافظه خطی مدیریت حافظه به عهده ی کامپایلر است و اگر برنامه ای برای مدیریت حافظه وجود داشته باشد معمولا در زبان بایت ماشین مجازی نوشته شده (مانند یک برنامه ی سی‌شارپ که به WASM ترجمه می شود). اگر حافظه غیر خطی باشد نیاز به GC[۸] و همچنین جدول نگاشت حافظه وجود دارد زیرا ماشین مجازی یک لایه‌ی انتزاعی بر روی صفحات مجازی حافظه سیستم عامل خواهد بود.

۲-۳ زمان‌بندی در ماشین های مجازی

چند راه حل برای زمان بندی کوتاه‌مدت در ماشین‌های مجازی وجود دارد. زمان بندی های یا پیشگیرانه اند، مانند زمان‌بندی در .BEAM یا مشارکتی (مانند Java و .(JavaScript در زمانبندی پیش‌گیرانه هر فرآیند فقط به شمارشی خاص از چرخه[۹] ها می تواند اجرا شود. چنین سیستم هایی می توانند به صورت خطی رشد کنند یعنی با اضافه شدن هسته های پردازنده ای که مفصر را اجرا می کند تعداد عملیات‌هایی که انجام می شوند به همان نسبت زیاد می شود. همچنین در صورت افزونی وضایف همگی به اندازه ی هم آهسته می شوند. چنین خسیسه ای برای اپلیکیشن هایی که وقت زیادی را در I/O می گذرانند و پردازش زیادی ندارند بسیار مناسب است. به همین خاطر است که مخابرات دنیا اکثرا با زبان ارلنگ نوشته شده. در زمان بندی های مشارکتی، نقطه ای باید وجود داشته باشد که پردازنده را به ماشین پس بدهد، وگرنه یک وظیفه خاص می تواند برای همیشه پردازنده را تصاحب کند. در Java تمام شدن هر تابع یا یک دور حلقه چنینی نقطه ای است. در JavaScript رسیدن به یک عملیات I/O چنین نقطه ای است.

[7] Heap
[8] Garbage Collection
[9] Cycle

۲-۴ ورودی/خروجی در ماشین‌های مجازی

در ماشین های مجازی ورودی/خروجی ها به صورت مسدود کننده و غیر مسدود کننده پیاده سازی می شوند. چون ورودی/خروجی در سیستم عامل های قدیمی مسدود کننده بوده ماشین‌های قدیمی مانند جاوا از ورودی/خروجی مسدود کننده پشتیبانی می کنند. البته با تمام شدن پروژه ی Loom جاوا نیز از ورودی/خروجی غیر مسدود کننده پشتیبانی خواهد کرد. مسدود کننده بودن و نبودن ورودی/خروجی به زمان‌بندی ماشین مجازی وابسته است. اگر ماشین مجازی از زمان بندی مشارکتی استفاده کند، برای ورودی/خروجی غیر مسدود کننده باید از رشته‌های سبز[۱۰] استفاده کند. در غیر این صورت باید از یک ماشین‌حالات[۱۱] برای مدیریت وضایف استفاده کند. async/await ای که در زبان‌هایی مثل سی‌شارپ و JavaScript می بینیم درواقع علامت هایی بای این ماشین‌حالات اند. هنگامی که یک عملیات به I/O می رسید همراه با وضعیتش به صف انتظار ماشین‌حالات رفته و ماشین از صف دیگری که مختص انجام شدگان است وظیفه ای را بر میدارد، وضعیت ذخیره شده ی آن را باز می گرداند و آن را ادامه می دهد. این آمد و شد ها همگی پشت صحنه توسط کامپایلر انجام می شود.

۲-۵ سطوح در ماشین‌های مجازی

ماشین های مجازی سطح بالا مختص یک زبان برنامه نویسی خاص هستند که به آن‌ها ماشین های سطح متوسط می گویند. پایتون و Perl این‌گونه اند و ماشین‌های مجازی آن دو عملیات های مختص همان زبان دارند. ماشین های مجازی دیگری مانند جاوا و CLR سطح پایین هستند و زبان بایتی آن‌ها را می توان (تقریبا) برای هر زبان برنامه نویسی ای استفاده کرد و آن‌ها را به کد سخت افزاری سریعی سروقت کامپایل کرد.

[10] Green Threads
[11] State Machine

فصل ۳

معرفی طرح

۳-۱ بررسی اجمالی

- زاگرس سامانه ای ۳۲ بیتی است
- زاگرس یک ماشین پشته ای است
- پشته‌ی داده آن ۳۲ عضو در خود جای می‌دهد
- پشته‌ی آدرس آن ۱۲۸ عضو در خود جای می‌دهد
- پیاده‌سازی مرجع، دسترسی به ۶۴ کیلو بایت حافظه خطی با دسترسی تصادفی را فراهم می‌کند
- همه‌ی عناصر پشته و ثبات‌های داخلی آن ۳۲ بیتی هستند
- ۸ هسته پردازش مرکزی مجازی در آن با زمان‌بندی نوبت گردشی[۱] دایر هستند
- هر هسته ۲۴ کلمه[۲] حافظه داخلی به عنوان ثبات، یک پشته‌ی داده، یک پشته‌ی آدرس، و ثبات‌های اشاره‌گر دستورالعمل[۳]، اشاره‌گر پشته[۴] و اشاره‌گر بازگشت[۵] را دارند.

[1] Round Robin
[2] Word
[3] Instruction Pointer
[4] Stack Pointer
[5] Return Pointer

- هسته‌ها می‌توانند فعال و غیر فعال شوند

۳-۲ حافظه

حافظه این سامانه از یک فضای آدرس تشکیل می‌شود. آدرس‌ها به صورت متوالی تعیین شده‌اند.

۳-۲-۱ آدرس دهی

حافظه می‌تواند با اندازه‌های بایت، نیم‌کلمه و کلمه آدرس‌دهی شود.

۳-۲-۲ تراز

کلمه‌ها باید با کرانه ۴ بایتی تراز شوند. نیازی نیست نیم کلمه‌ها و بایت‌ها هنگامی که به عنوان داده استفاده می‌شوند تراز شوند. برای خواندن یک بایت ۸ بیت اول مقدار نگاشته‌ی آدرس را می‌خوانیم و می‌نویسیم و باقی بیت‌ها را نادیده می‌گیریم. برای نمونه:

Table ۳-۱: حافظه تراز جدول

Address	۰-۷	۸-۱۵	۱۶-۲۳	۲۴-۳۱
۰۰۰۰	byte	half		word
۰۰۰۱	byte			
۰۰۱۰	byte	half		
۰۰۱۱	byte			
۰۱۰۰	byte	half		word
۰۱۰۱	byte			
۰۱۱۰	byte	half		
۰۱۱۱	byte			

۳-۲-۳ نگاشت حافظه

نخستین ۱۹۲ بایت حافظه برای محوطه واسط‌های I/O رزرو شده اند. ازینجا به بعد حافظه عموما در اختیار برنامه‌ها قرار دارد. با این حال بعضی دستگاه‌ها می‌توانند بخشی از فضای حافظه را برای I/O حافظه نگاشتی تخصیص دهند. دستگاه‌ها می‌توانند مناطق حافظه بیشتری نگاشت کنند که باید در پیاده سازی مستند و مطالعه شود.

۳-۳ پردازش دستورالعمل‌ها

زاگرس دستورالعمل‌ها را به عنوان بسته‌های کلمه اندازه پردازش می‌کند (اگر کلمه‌ها ۳۲ بیتی باشند، دستورالعمل‌ها ۳۲ بیت ۳۲ بیت واکشی می‌شوند). هر بسته می‌تواند تا چهار جایگاه دستورالعمل و داده‌ی مربوط داشته باشد. دستورالعمل‌ها و اطلاعات آن باید در یک بسته جا شوند.

بسته‌های دستورالعمل باید در کرانه‌های کلمه ای تراز شوند. برای مثال می‌توانیم دنباله ای از بسته‌ها به شکل زیر داشته باشیم:

```
LDB 1
ADD
RET

DUP
MUL
RET
...
```

به شرط اینکه مولفه‌های برخط[۶] در یک کلمه باشند. کد زیر درست خواهد بود:

```
DUP
MUL
```

[6] Inline

```
LDB 45
```

در حالی که کد زیر خیر:

```
DUP
MUL
LDH 45
```

دستور LDH (بخوانیم load (half یک نیم کلمه را بارگذاری می‌کند پس مولفه خطی آن یک مقدار ۱۶ بیتی در نظر گرفته می‌شود. که مجموعه این سه خط دستور را ۴۰ بیت می‌کند که از ۳۲ بیت کلمه بیشتر است.

موارد خاص دستور LDW مقدار کلمه موجود در بسته بعدی را در پشته push می‌کند. دستور LDH مقدار نیم کلمه (۱۶ بیت) موجود در جایگاه دستورالعمل بعدی را در پشته push می‌کند. دستور LDB مقدار بایت (۸ بیت) موجود در جایگاه دستورالعمل بعدی را در پشته push می‌کند. دستورهای CAL CNC، JUM، CNJ، RET، CNR، و TRI اشاره گر دستورالعمل (IP) را تغییر می‌دهند و نمی توانند توسط هیچ دستورالعملی به جز NOP دنبال شوند. هر دستورالعملی که بعد از آن‌ها خطی شود توسط زاگرس به عنوان NOP تلقی می‌شود.

دستورالعمل‌ها در یک کلمه ۳۲ بیتی بسته‌بندی می‌شوند و به چهار جایگاه ۸ بیتی تقسیم می‌شوند.

دستور lh می‌تواند در جایگاه ۰ و ۱ استفاده شود. چون به دنبال آن مقداری ۱۶ بیتی (۲ جایگاهی) می‌آید که نباید از کران یک کلمه ی بسته ی دستورالعملش خارج شود. در غیر این صورت مولفه ی خطی (داده ی) بیرون از بسته دستورالعمل قرار می‌گیرد. lb می‌تواند در جایگاه‌های ۰، ۱ و ۲ قرار گیرد چون به دنبال آن یک مقدار ۸ بیتی می‌آید.

هر دستورالعملی که به دنبال دستورالعمل‌هایی بیاید که اشاره گر دستورالعمل (IP) را تغییر می‌دهند نادیده گرفته می‌شود (یا به عنوان NOP تلقی می‌شوند). این دستورالعمل‌ها CNJ، RET، CNR، JUM، CAL CNC و TRI هستند.

اشاره گر دستورالعمل (IP) بعد از پردازش یافتن بسته ۱ واحد افزایش می‌یابد.

حافظه می تواند مقادیر ۳۲ بیتی، ۱۶ بیتی، و ۸ بیتی را واکشی و ذخیره سازی کند. اگر آدرسی که به این توابع داده می شود غیر صحیح باشد وقفه ی مربوط به آن انداخته می شود.

۳-۴ پشته

پشته ها از دو دستورالعمل پاپ و پوش پشتیبانی می کنند و اگر سرریز یا ته‌ریز داشته باشند وقفه های مربوط به خطایشان را می اندازند.

۳-۵ محفظه ثبات ها

محفظه ی ثبات ها می تواند ثبات ها را بخواند و در آن‌ها بنویسد. اگر شماره ثباتی که به این توابع داده می شود غیر صحیح باشد وقفه ی مربوط به آن انداخته می شود.

۳-۶ جدول ورودی/خروجی

ماشین مجازی می تواند توابعی از زبان میزبان را به عنوان دستگاه دریافت کند و ورودی/خروجی آن را انجام دهد. اگر شماره ورودی/خروجی که به این توابع داده می شود غیر صحیح باشد وقفه ی مربوط به آن انداخته می شود. دستگاه‌ها می توانند در بخشی از حافظه که مربوط به آن‌هاست دسترسی داشته باشند، یا حتی حافظه می تواند جوری تنظیم شود که با خواندن و نوشتن در یک آدرس خاص، ورودی/خروجی انجام دهد. اطلاعات مربوط به سیستم می تواند به صورت پیش‌فرض توابعی در این جدول باشد.

۳-۷ جدول وقفه‌ها

ماشین مجازی می تواند آدرس توابع مربوط به هر شماره وقفه را در خود نگه دارد و در صورت وقوع وقفه آن آدرس را فراخوانی کند. گر شماره وقفه که به این توابع داده می شود غیر صحیح باشد وقفه ی مربوط به آن انداخته می شود. وقفه ها می توانند غیر فعال شوند یا بردار (وصل کننده شماره وقفه به آدرس) شان تعویض شود.

۳-۸ دستورالعمل ها

زاگرس در حال حاضر از ۵۳ دستورالعمل پشتیبانی می کند، اما در فرمت دنباله بایتی خود تا ۲۵۶ دستور را زرور می کند. تغییرات دستورالعمل ها با S-expression نوشته شده. تلاش شده تا الگوریتم ها به خوانا و آشناترین حالت ممکن نوشته شوند.

۳-۸-۱ NOP

- کد: ۰۰
- شرح: عملیات خالی.
- تلفظ انگلیسی: No-Op
- تغییرات:

```
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

۳-۸-۲ LDW

- کد: ۰۱
- شرح: مقداری ۳۲ بیتی را در پشته پوش می‌کند.
- تلفظ انگلیسی: Load Word
- تغییرات:

```
(push data-stack (get-word memory (+ 4 instruction-pointer)))
(increment instruction-pointer boundle-pointer 8)
```

LDH ۳-۸-۳

- کد: ۰۲
- شرح: مقداری ۱۶ بیتی را در به ۳۲ بیت تبدیل کرده به پشته پوش می‌کند.
- تلفظ انگلیسی: Load Half
- تغییرات:

```
(push data-stack (get-half memory (+ 1 instruction-pointer)))
(increment instruction-pointer boundle-pointer 3)
```

LDB ۴-۸-۳

- کد: ۰۳
- شرح: مقداری ۸ بیتی را به ۳۲ بیت تبدیل کرده در پشته پوش می‌کند.
- تلفظ انگلیسی: Load Byte
- تغییرات:

```
(push data-stack (get-byte memory (+ 1 instruction-pointer)))
(increment instruction-pointer boundle-pointer 2)
```

FEW ۵-۸-۳

- کد: ۰۴
- شرح: مقداری ۳۲ بیتی را از حافظه واکشی می کند.
- تلفظ انگلیسی: Fetch Word
- تغییرات:

```
(push data-stack (get-word memory (pop data-stack)))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

FEH ۶-۸-۳

- کد: ۰۵
- شرح: مقداری ۱۶ بیتی را به عنوان یک مقدار ۳۲ بیتی از حافظه واکشی می کند.
- تلفظ انگلیسی: Fetch Half
- تغییرات:

```
(push data-stack (get-half memory (pop data-stack)))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

FEB ۷-۸-۳

- کد: ۰۶
- شرح: مقداری ۱۶ بیتی را به عنوان یک مقدار ۳۲ بیتی از حافظه واکشی می کند.
- تلفظ انگلیسی: Fetch Byte
- تغییرات:

```
(push data-stack (get-byte memory (pop data-stack)))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

STW ۸-۸-۳

- کد: ۰۷

- شرح: مقداری ۳۲ بیتی را در حافظه ذخیره می کند.
- تلفظ انگلیسی: Store Word
- تغییرات:

```
(set-word memory (pop data-stack) (pop data-stack))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

STH ۹-۸-۳

- کد: ۰۸
- شرح: مقداری ۱۶ بیتی را در حافظه ذخیره می کند.
- تلفظ انگلیسی: Store Half
- تغییرات:

```
(set-half memory (pop data-stack) (pop data-stack))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

STB ۱۰-۸-۳

- کد: ۰۹
- شرح: مقداری ۸ بیتی را در حافظه ذخیره می کند.
- تلفظ انگلیسی: Store Byte
- تغییرات:

```
(set-byte memory (pop data-stack) (pop data-stack))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

DUP ۱۱-۸-۳

- کد: A۰
- شرح: مقدار بالای پشته را تکثیر می کند.
- تلفظ انگلیسی: Duplicate
- تغییرات:

```
(let [val (pop data-stack)]
  (push data-stack val)
  (push data-stack val))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

DRP ۱۲-۸-۳

- کد: B۰
- شرح: مقدار بالای پشته را رها می کند.
- تلفظ انگلیسی: Drop
- تغییرات:

```
(pop data-stack)
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

SWP ۱۳-۸-۳

- کد: C۰
- شرح: دو مقدار بالای پشته را باهم تعویض می کند.

- تلفظ انگلیسی: Swap
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push data-stack second)
(push data-stack first))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

PUA ۱۴-۸-۳

- کد: D۰
- شرح: مقدار بالای پشته را به پشته‌ی آدرس پوش می کند.
- تلفظ انگلیسی: Push Address
- تغییرات:

```
(push address-stack (pop data-stack))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

POA ۱۵-۸-۳

- کد: E۰
- شرح: مقدار بالای پشته‌ی آدرس را پاپ کرده به پشته پوش می کند.
- تلفظ انگلیسی: Pop Address
- تغییرات:

```
(push data-stack (pop address-stack))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

EQL ۱۶-۸-۳

- کد: F۰
- شرح: برابری دو مقدار را مقایسه می کند.
- تلفظ انگلیسی: Equal
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (= first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

NEQ ۱۷-۸-۳

- کد: ۱۰
- شرح: نابرابری دو مقدار را مقایسه می کند.
- تلفظ انگلیسی: Not Equal
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (!= first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

LET ۱۸-۸-۳

- کد: ۱۱

- شرح: کوچکتری دو مقدار را مقایسه می کند.
- تلفظ انگلیسی: Less than
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (< first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

GRT ۱۹-۸-۳

- کد: ۱۲
- شرح: بزرگتری دو مقدار را مقایسه می کند.
- تلفظ انگلیسی: Greater than
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (> first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

ADD ۲۰-۸-۳

- کد: ۱۳
- شرح: دو مقدار بالای پشته را باهم جمع می کند.
- تلفظ انگلیسی: Add

- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (+ first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

SUB ۲۱-۸-۳

- کد: ۱۴
- شرح: دو مقدار بالای پشته را از هم کم می کند.
- تلفظ انگلیسی: Subtract
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (- first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

MUL ۲۲-۸-۳

- کد: ۱۵
- شرح: دو مقدار بالای پشته را در هم ضرب می کند.
- تلفظ انگلیسی: Multiply
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (* first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

DVR ۲۳-۸-۳

- کد: ۱۶
- شرح: دو مقدار بالای پشته را بر هم تقسیم می کند. اگر مقسوم علیه ۰ باشد dividie-by-zero-interrupt انداخته می شود. باقی‌مانده و خارج‌قسمت را بر روی پشته پوش می کند.
- تلفظ انگلیسی: Remainder and Divide
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (/ first second))
(push (% first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

MDR ۲۴-۸-۳

- کد: ۱۷
- شرح: دو مقدار بالای پشته را در هم ضرب می کند و حاصل را بر مقدار سوم پشته تقسیم می کند. اگر مقسوم علیه ۰ باشد dividie-by-zero-interrupt انداخته می شود. باقی‌مانده و خارج‌قسمت را بر روی پشته پوش می کند.
- تلفظ انگلیسی: Remainder and Divide Multiply،

- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)
      third  (pop data-stack)]
(push (/ (* first second) third))
(push (% (* first second) third)))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

AND ۲۵-۸-۳

- کد: ۱۸
- شرح: دو مقدار بالای پشته را و باینری می کند.
- تلفظ انگلیسی: And
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (& first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

IOR ۲۶-۸-۳

- کد: ۱۹
- شرح: دو مقدار بالای پشته را یا باینری می کند.
- تلفظ انگلیسی: Inclusive Or
- تغییرات:

```
(let [first  (pop data-stack)
second (pop data-stack)]
(push (| first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

XOR ۲۷-۸-۳

- کد: A۱
- شرح: دو مقدار بالای پشته را با انصحاری باینری می کند.
- تلفظ انگلیسی: Exclusive Or
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (^ first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

NOT ۲۸-۸-۳

- کد: B۱
- شرح: مقدار بالای پشته را نفی می کند.
- تلفظ انگلیسی: Not
- تغییرات:

```
(push data-stack (~ (pop data-stack)))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

SHL ۲۹-۸-۳

- کد: C۱
- شرح: مقدار اول بالای پشته را به مقدار دوم بالای پشته شیفت چپ می دهد.
- تلفظ انگلیسی: Shift Left
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (<< first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

SHL ۳۰-۸-۳

- کد: D۱
- شرح: مقدار اول بالای پشته را به مقدار دوم بالای پشته شیفت راست می دهد.
- تلفظ انگلیسی: Shift Right
- تغییرات:

```
(let [first  (pop data-stack)
      second (pop data-stack)]
(push (>> first second))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

PAC ۳۱-۸-۳

- کد: E۱

- شرح: از چهار مقدار بالای پشته کم ارزش ترین بایت ها را برداشته و تبدیل به یک مقدار ۳۲ بیتی می کند.
- تلفظ انگلیسی: Pack
- تغییرات:

```
(push data-stack
  (|
    (<< (byte (pop data-stack)) 24)
    (<< (byte (pop data-stack)) 16)
    (<< (byte (pop data-stack)) 08)
    (<< (byte (pop data-stack)) 00)))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

UNP ۳۲-۸-۳

- کد: F۱
- شرح: مقدار بالای پشته را برداشته و هر یک از بایت های آن را تبدیل به یک مقدار ۳۲ بیتی می کند.
- تلفظ انگلیسی: Unpack
- تغییرات:

```
(let [val (pop data-stack)]
    (push data-stack (& 0xFF (<< val 00)))
    (push data-stack (& 0xFF (<< val 08)))
    (push data-stack (& 0xFF (<< val 16)))
    (push data-stack (& 0xFF (<< val 24))))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

REL ۳۳-۸-۳

- کد: ۲۰
- شرح: فراخوانی بعدی را به صورت نسبی در نظر می گیرد.
- تلفظ انگلیسی: Relative
- تغییرات:

```
(set relative true)
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

CAL ۳۴-۸-۳

- کد: ۲۱
- شرح: اگر در حالت نسبی باشیم با مقدار بالای پشته فراخوانی نسبی و اگر نه فراخوانی مستقیم انجام می دهد. آدرس دستورالعمل بعدی را در پشته‌ی آدرس پوش می کند.
- تلفظ انگلیسی: Call
- تغییرات:

```
(push address-stack (increment instruction-pointer boundle-pointer 4))
(if relative
    (set instruction-pointer (+ (pop data-stack) (instruction-pointer)))
    (set instruction-pointer (pop data-stack)))
```

CNC ۳۵-۸-۳

- کد: ۲۲

- شرح: در صورت صحیح بودن شرط مقدار اول پشته، با مقدار دوم بالای پشته اگر در حالت نسبی باشیم فراخوانی نسبی و اگر نه فراخوانی مستقیم انجام می دهد. آدرس دستورالعمل بعدی را در پشته‌ی آدرس پوش می کند.
- تلفظ انگلیسی: Conditional Call
- تغییرات:

```
(if (pop data-stack)
    (do
        (push address-stack (increment instruction-pointer boundle-pointer 4))
        (if relative
            (set instruction-pointer (+ (pop data-stack) (instruction-pointer)))
            (set instruction-pointer (pop data-stack)))
    (set instruction-pointer (increment instruction-pointer boundle-pointer 4))))
```

JUM ۳۶-۸-۳

- کد: ۲۳
- شرح: اگر در حالت نسبی باشیم با مقدار بالای پشته پرش نسبی و اگر نه پرش مستقیم انجام می دهد.
- تلفظ انگلیسی: Jump
- تغییرات:

```
(if relative
    (set instruction-pointer (+ (pop data-stack) (instruction-pointer)))
    (set instruction-pointer (pop data-stack)))
```

CNJ ۳۷-۸-۳

- کد: ۲۴
- شرح: در صورت صحیح بودن شرط مقدار اول پشته، با مقدار دوم بالای پشته اگر در حالت نسبی باشیم پرش نسبی و اگر نه پرش مستقیم انجام می دهد.
- تلفظ انگلیسی: Conditional Jump
- تغییرات:

```
(if (pop data-stack)
    (do
       (if relative
         (set instruction-pointer (+ (pop data-stack) (instruction-pointer)))
          (set instruction-pointer (pop data-stack)))
    (set instruction-pointer (increment instruction-pointer boundle-pointer 4))))
```

RET ۳۸-۸-۳

- کد: ۲۵
- شرح: مقدار بالای پشته‌ی آدرس را پاپ کرده و به آدرس آن باز می گردد.
- تلفظ انگلیسی: Return
- تغییرات:

```
(set instruction-pointer (pop address-stack))
```

CNR ۳۹-۸-۳

- کد: ۲۶

- شرح: در صورت صحیح بودن شرط مقدار اول پشته، مقدار بالای پشته‌ی آدرس را پاپ کرده و به آدرس آن باز می گردد.
- تلفظ انگلیسی: Conditional Return
- تغییرات:

```
(if (pop data-stack)
  (set instruction-pointer (pop address-stack))
  (set instruction-pointer (increment instruction-pointer boundle-pointer 4)))
```

SIV ۴۰-۸-۳

- کد: ۲۷
- شرح: بردار وقفه را تنظیم می کند.
- تلفظ انگلیسی: Set Interrupt Vector
- تغییرات:

```
(set-bit interrupt-vector-table (pop data-stack) (pop data-stack))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

HLI ۴۱-۸-۳

- کد: ۲۸
- شرح: وقفه ها را غیر فعال می کند.
- تلفظ انگلیسی: Halt Interrupts
- تغییرات:

```
(set interrupts-enabled? false)
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

STI ۴۲-۸-۳

- کد: ۲۹
- شرح: وقفه ها را فعال می کند.
- تلفظ انگلیسی: Start Interrupts
- تغییرات:

```
(set interrupts-enabled? false)
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

TRI ۴۳-۸-۳

- کد: A۲
- شرح: یک وقفه را راه اندازی می کند.
- تلفظ انگلیسی: Trigger Interrupts
- تغییرات:

```
(if interrupts-enabled?
    (do
        (push address-stack instruction-pointer))
        (set instruction-pointer (get interrupt-table (pop data-stack))))
    (set instruction-pointer (increment instruction-pointer boundle-pointer 4)))
```

IIO ۴۴-۸-۳

- کد: B۲

- شرح: با مقدار بالای پشته به عنوان شماره ی دستگاه، عملیات ورودی/خروجی آن را انجام می دهد.
- تلفظ انگلیسی: Invoke I/O
- تغییرات:

```
((get io-table (pop data-stack)))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

HLS ۴۵-۸-۳

- کد: C۲
- شرح: سامانه را متوقف می کند.
- تلفظ انگلیسی: Halt System
- تغییرات:

```
(throw :system-halt-interrupt)
```

INC ۴۶-۸-۳

- کد: D۲
- شرح: اشاره‌گر برنامه ی هسته ی خواسته شده را تنظیم می کند.
- تلفظ انگلیسی: Initialize Core
- تغییرات:

```
(let [core (get cores (pop data-stack))
        core-ip (pop data-stack)]
    (set (get core :instruction-pointer) core-ip))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

ACC ۴۷-۸-۳

- کد: E۲
- شرح: هسته ی خواسته شده را فعال می کند.
- تلفظ انگلیسی: Activate Core
- تغییرات:

```
(let [core (get cores (pop data-stack))]
    (set (get core :active) true))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

PAC ۴۸-۸-۳

- کد: F۲
- شرح: هسته ی خواسته شده را وادار به مکث می کند.
- تلفظ انگلیسی: Pause Core
- تغییرات:

```
(let [core (get cores (pop data-stack))]
(set (get core :active) false))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

SCC ۴۹-۸-۳

- کد: ۳۰
- شرح: هسته ی فعلی را تعلیق می کند.

- تلفظ انگلیسی: Suspend Current Core
- تغییرات:

```
(let [core (get cores current-core-id)]
(set (get core :active) false))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

RER ۵۰-۸-۳

- کد: ۳۱
- شرح: ثبات خواسته شده را می خواند و در پشته پوش می کند.
- تلفظ انگلیسی: Read Register
- تغییرات:

```
(push data-stack (get register-bank (pop data-stack)))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

WRR ۵۱-۸-۳

- کد: ۳۲
- شرح: مقدار بالای پشته را در ثبات خواسته شده می نویسد.
- تلفظ انگلیسی: Write Register
- تغییرات:

```
(set register-bank (pop data-stack) (pop data-stack))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

CPB ۵۲-۸-۳

- کد: ۳۳
- شرح: تعداد کلمات خواسته شده را از آدرس مقصد به آدرس مبدا کپی می کند. این دستور مانند فراخوانی‌سیستمی memcpy در زبان C است و احتمال override شدن وجود دارد.
- تلفظ انگلیسی: Copy Block
- تغییرات:

```
(let [destination (pop data-stack)
      length      (pop data-stack)
      origin      (pop data-stack)]
          (copy
              (+ (begin memory) origin)
              length
              (+ (begin memory) destination)))
        (set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

CMB ۵۳-۸-۳

- کد: ۳۴
- شرح: تعداد کلمات خواسته شده را از آدرس مقصد با آدرس مبدا مقایسه می کند. -۱ به نشانه ی کوچکتری، ۰ به نشانه برابری و ۱ به نشانه ی بزرگتری در پشته پوش می شود.
- تلفظ انگلیسی: Compare Block
- تغییرات:

```
(let [destination (pop data-stack)
      length      (pop data-stack)
```

```
      origin      (pop data-stack)]
    (equal
        (+ (begin memory) destination)
        (+ (begin memory) destination length)
        (+ (begin memory) origin)))
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

UNS ۵۴-۸-۳

- کد: ۳۵
- شرح: دستورالعمل ریاضی بعدی به صورت بی علامت انجام می شود.
- تلفظ انگلیسی: Unsigned
- تغییرات:

```
(set unsigned true)
(set instruction-pointer (increment instruction-pointer boundle-pointer 1))
```

۹-۳ پردازش موازی

زاگرس از زمان‌بندی نوبت گردشی استفاده می کند. یعنی به نوبت از هسته ی شماره ۰ تا ۷، در صورتی که هسته فعال باشد، فقط یک دستور اجرا می شود و نوبت به هسته‌ی بعدی داده می‌شود تا آن هم یک دستور انجام دهد.

۱۰-۳ ماشین

ماشین با خواندن خانه ی اول حافظه تفسیر را شروع می کند و تا رسیدن به وقفه ی توقف سامانه کار خود را ادامه می دهد. برای استفاده از ماشین به عنوان پوسته می توان از تنظیمات ورودی/خروجی

استفاده کرد. توابعی برای گرفتن snapshot لحظه ای از ماشین ارائه داده شده اند اما این توابع خالص[۷] هستند و تغییری در وضعیت ماشین ایجاد نمی کنند.

[7] Pure

فصل ۴

پیاده‌سازی

۴-۱ زبان پیاده سازی

زاگرس ابتدا با ++C۲۰ پیاده سازی شد. اما بعد ها برا پرتابل تر شدن و قابلیت استفاده شدن از آن در معماری های غیر مرسوم و سامانه‌های‌نفته به ++C۱۱ ری‌فکتور شد. در حال حاضر زاگرس در ۵۹۴۳ خط کد نوشته شده و تقریبا همه ی توابع و شاخه های آن مستند سازی شده.

۴-۲ آزمایش

تمامی instruction ها و توابع عمومی زاگرس و ساختمان داده هایی که استفاده می کند test unit شده. از فریمورک googletest و ۱۷۷۳ خد کد تست دارد.

۴-۳ قابلیت پیکربندی شدن

زاگرس ابتدا از طریق template های C++ پیکر بندی شده و منابع آن تنظیم می شد. اما الان مانند پروژه‌های نفته در زبان C از طریق macro def ها پیکربندی می شود.

۴-۴ محدودیت های پیاده‌سازی

در پیاده سازی زاگرس از حافظه‌ی Heap استفاده نشده. تخصیص از این بخش از حافظه هیچگاه تضمین‌شده نیست به همین خاطر در توسعه‌ی نهفته از آن استفاده نمی شود. حافظه‌ی Heap از محیط اجرای برنامه دورتر است پس دسترسی به آن آهسته‌تر است. به علاوه هر بار دسترسی به این حافظه ممکن است حافظه‌ی پنهان پردازنده را باطل کند. اما اگر برنامه فقط در فضای حافظه پشته وجود داشته باشد، می تواند به شرط اینکه از فضای کمی استفاده کند در حافظه‌ی پنهان بماند و هیچگاه نیازی به استفاده از حافظه‌ی RAM نباشد. زاگرس با پرچم noexcept و nortti کامپایل شده. استفاده نکردن از exception ها فواید زیادی دارد اما در مورد زاگرس، سایز کد تولید شده توسط کامپایلر را به شدت کاهش می دهند و به ما کمک می کنند اندازه‌ی برنامه را کاهش دهیم. استفاده نکردن از runtime information type ها باعث کاهش مصرف حافظه می شود.

۴-۵ نحوه تفسیر

زاگرس از تفسیر رشته‌ایInterpretation LTRfootnoteThreaded استفاده می کند. پردازنده های مدرن بهینه سازی شده اند تا پرفورمنس خوبی در اعزام‌مجازی[1] داشته باشند. بهینگی دستورالعمل‌های پرش دستورالعمل های شاخه‌سازی[2] بسیار بیشتر است زیرا در بدترین حالت شاخه‌سازی باید همگی شاخه ها بررسی شوند. در پردازنده های قوی‌تر که اجرای خارج از نوبت[3] دارند تفاوت این بهینگی هنوز هم بیشتر است. و در نهایت چون زاگرس تلاش می کند در حافظه‌ی پنهان زندگی کند تاثیر این بهینگی هنوز هم بیشتر است. در مفسر های رشته‌ای، آدرس تفسیر هر دستورالعمل در یک جدول قرار دارد. اشاره‌گر برنامه همواره نقش اندیس این جدول را خواهد داشت. هر دستورالعمل اشاره‌گر را برای دستورالعمل بعدی آماده می کند. برای پیاده‌سازی همچین عملی نیاز به افزونه‌ی goto computed در زبان C/C++ است که هم GCC و هم clang از آن پشتیبانی می کنند.

[1] Virtual Dispatch
[2] Branching
[3] Out of order execution

۴-۶ بهینه‌سازی های دستی

کد ترجمه شده ی کامپایلر برای زاگرس به صورت دستی برای معماری‌های ۶۴AMD و ۶۴ARM بررسی شده تا مطمئن شویم انتزاعات C++ تاثیر منفی ای در بهینگی نداشته باشد، استفاده از کتابخانه ها دستورالعمل های سخت‌افزاری ناخواسته تولید نکرده باشد، و توابع مربوط به تفسیر رشته ای همگی برخط شده باشند. زاگرس در حال حاضر با استفاده از پرچم O-۳ کمتر از ۱۸۰۰ دستورالعمل سخت‌افزاری در هر دو معماری دارد.

۴-۷ استفاده از SWIG

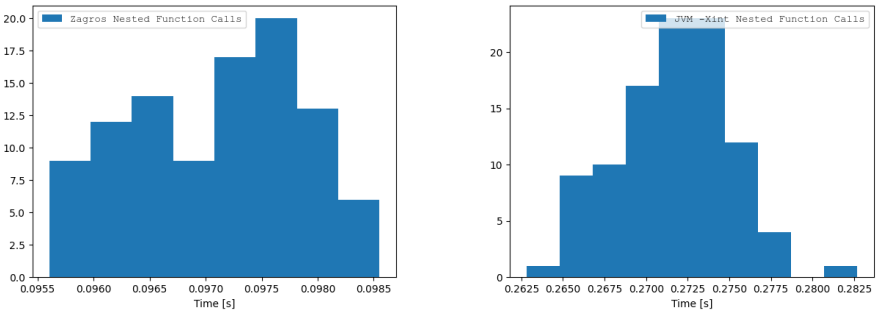
با استفاده از SWIG می‌توان برای اکثر زبان های مطرح دنیا برای زاگرس اتصالات تولید کرد و از زاگرس و بهینگی C++ در آن زبان ها استفاده کرد. حتی ورودی/خروجی‌ها را می توان در زبان مقصد نوشت.

فصل ۵

نتیجه‌گیری

۵-۱ بهنیگی فراخوانی توابع

زاگرس چون اطلاعات بسیار کمتری از پشته ردیابی می کند و همینطور چون در پیاده‌سازی مرجع فقط از ۱۲۸ فراخوانی تو در تو پشتیبانی می کند می‌تواند فراخوانی به و بازگشت از توابع را با سرعتی تقریبا ۳ برابر از مفسر جاوا انجام دهد.

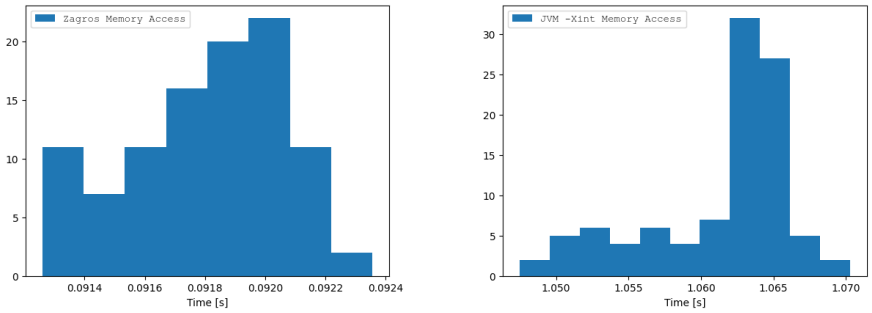


(ب) زاگرس (آ) مفسر جاوا

شکل ۵-۱: آزمایش بهینگی فراخوانی تودرتو توابع

۵-۲ بهینگی دسترسی به حافظه

زاگرس چون حافظه‌ای خطی دارد و هیچ داد و ستدی با سیستم عامل در مدیریت حافظه انجام نمی دهد دسترسی خواندن و نوشتنش تقریبا ۱۰ برابر سریع تر از مفسر جاوا است. در قبال این مزایا زاگرس قابلیت حافظه‌پویا[1] ندارد و برنامه‌نویس و کامپایر باید مانند هر برنامه ی Embedded دیگری همه‌ی حافظه‌ی مورد نیاز خود را به صورت ایستا تهیه کند. همینطور میزان حافظه‌ی پشتیبانی شده در پیاده‌سازی مرجع فقط ۶۴ کیلوبایت است.



(آ) مفسر جاوا (ب) زاگرس

شکل ۵-۲: آزمایش بهینگی دسترسی به حافظه

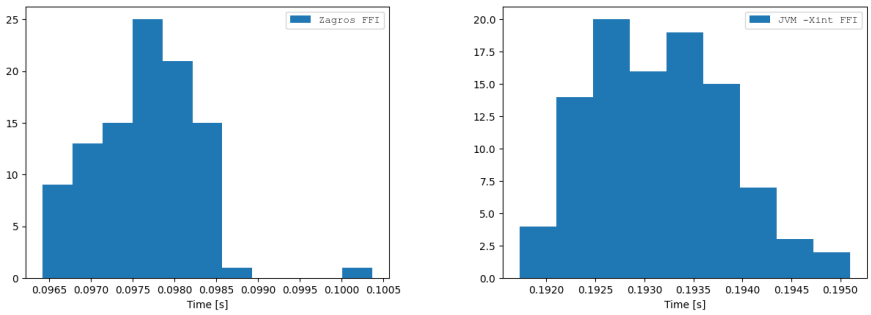
۵-۳ بهینگی فراخوانی توابع سیستمی

زاگرس چون با C++ نوشته شده، نیازی به لایه‌های محافظتی ندارد و سربار کمتری برای صدا زدن توابع سیستمی دارد. پس تقریبا در FFI[2] ها ۲ برابر از مفسر جاوا سریعتر است.

۵-۴ نهایت

وقتی امکانات مورد نیاز ما برای پیاده‌سازی یک زبان اندک است، مانند هنگامی که یک زبان مختص دامنه پیاده سازی می کنیم، می توانیم با استفاده از ماشین‌های مجازی سبک‌تر و اختصاصی‌تر مصرف

[1] Dynamic Memory
[2] Foregin Function Interface



(آ) مفسر جاوا (ب) زاگرس

شکل ۵-۳: آزمایش بهینگی فراخوانی توابع سیستمی

منابع کمتر و بهینگی بیشتری داشته باشیم.

پیوست آ

گیت‌هاب

https://github.com/daeshti/zagros

مراجع

واژەنامه

Abstract

Zagros is an embeddable virtual computing system. It is lightweight enough to be used in embedded systems and performant enough to be used by server applications. The hardware resources it abstracts are all configurable so it can be build for different needs. It uses SWIG to create interfaces for most popular programming languages. Through bindings it can provide a reactive shell, live upgrades, support real-time programming and upgrading, and so on.

Keywords: Virtual Machines, Embedded Systems, Live Update

Persian Gulf University

Department of Data Science and Intelligent Systems

B.Sc. Thesis

Zagros Virtual Machine

By:

Arian Dashti

Supervisor:

Dr. Sahafi-Zade

September 2022