دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

گزارش کار پروژه

درس ساختار زبان و ماشین

استاد: دکتر حسین اسدی

دانشجویان:

سید محمد آقامیر، سید مهران خلدی، محمد حسین سخاوت، سجاد جلالی، محمّدرضا کسنوی

تیر ۱۳۹۱

فهرست

۱- معرفی پروژه

در این پروژه سعی کردیم الگوریتم کدینگ هافمن را توسط MIPS Assembly پیاده‌سازی کنیم.

۱-۱- خلاصه: چرا هافمن؟

می‌دانیم که در حالت عادی، هر کاراکتر از یک فایل متنی توسط کد اسکی آن ذخیره می‌شود. این کد برای تمامِ کاراکترها ۸-بیتی است. از سوی دیگر، می‌دانیم که حرف q بسیار کمتر از حرف e در متون کاربرد دارد. پس منطقی به‌نظر می‌رسد که e دارای کد کوتاهتری نسبت به q باشد. الگوریتم هافمن تلاش می‌کند تا هر حرف (و در حالت کلّی‌تر سیمبل) را با یک کد نمایش دهد، به نحوی که تعداد کلّ بیت‌هایی که برای ذخیره‌سازی متن مورد نیاز است کمینه باشد. به همین سادگی!

۱-۲- آشنایی با هافمن

در مبحث مهندسی کامپیوتر و نظریه الگوریتم هافمن نوعی (entropy algorithm) که آن دسته ای از الگوریتم ها برای ترجمه اطلاعات فشرده شده و رمزنگاری شده می باشد که به نوع کاراکتر های استفاده شده و تعداد آنها مربوط می باشد که در اکثر اوقات به این صورت می باشد که تعداد استفاده شده از هر کاراکتر را در خود نگه می دارد تا با استفاده از آن و احتمال حضور کاراکتر مورد نظر، در رمز نگاری به جواب درست برسد که برای این دسته از الگوریتم ها بهترین و اپتیمال شده ترین اندازه کد، از اردر $\log_b^P$ می باشد که در آن b تعداد تنوع کاراکتر ها و P برابر احتما حضور کاراکتر(symbol) مورد نظر می باشد. و این نظریه به نظریه منبع رمزنگاری شانن (Shannon’s source coding theory) معروف می باشد. که Huffman coding از معروف ترین انواع آن می باشد.

نظریه هافمن، برای هر عنصر احتمال حضور خاصی را در نظر می گیرد. و به این ترتیب سعی در فشرده سازی و رمز نگاری اطلاعات می کند. و بر این اساس درختی طراحی و رسم می کند تا با استفاده از آن عملیات رمز نگاری خود را انجام، دهد، که در آن درخ، بر اساس بیتهای ورودی می توان روی آن پیمایش کرد و هر عنصر ورودی را شناسایی و جایگزاری کرد.

این نظریه توسط David A. Huffman از دانشگاه MIT به عنوان دانشجوی دکتری و در سال ۱۹۵۲ منتشر شده است.به عنوان "ساختاری بر اساس کمترین تعداد و طول کد برای کاهش طول اطلاعات" نامگزاری شده است. نظریه او به این صورت بود که طول متن را به ساختاری از بیت ها تبدیل می کنیم که به این صورت که هر کاراکتری که احتمال حضور بیشتری داشته

باشد، از تعداد بیت کمتری برای نمایش آن استفاده می کنیم. همچنین بعدها نشان داده شد که اگر احتمال حضور هر کاراکتر را به صورت دسته بندی شده داشته باشیم، این روش را می توان در زمان خطی پیاده سازی کرد.

اما در پیاده سازی این رمز نگاری برای دسته ای از کاراکتر ها با احتمال حضور برابر، قابلیت پوشاندن این دسته در دسته n تایی از کاراکتر ها به صورت استفاده log n بیت برای هر کاراکتر انجام می شود که این الگوریتم، الگوریتمی معمول و شناخته شده برای ذخیره اطلاعات که به binary block encoding معروف می باشد نام دارد. این موضوع از ضعفهای این روش می باشد که در جایی که احتمال حضور برابر باشد، قدرت فشرده سازی این روش زیر سوال می رود، ولی با این حال اندازه آن از حالت طبیعی خارج نیست و هنوز می توان آن را به صورت روشی برای رمز نگاری استفاده کرد. همچنین مسئله دیگری که این روش را نسبت به روش هایی مانند arithmetic coding یا LZW ضعیف تر نشان می دهد مسئله اهمیت ندادن به احتمال حضور در کلمات است، مثلا در نظر بگیرید، در متنی فارسی، احتما حضور کلمه ای مانند درخت خیلی بیشتر از کلمه ای مانند درخ می باشد، پس می توان تعداد بیت کمتری برای آن در نظر گرفت تا از حجم کد بکاهد.

با این وجود، این الگوریتم مانند الگوریتم های بزرگی مثل ASCII coding الگوریتمی شناخته شده می باشد که در بسیاری از الگوریتم ها و روشهای ذخیره سازی اطلاعات، بسیار شناخته شده و معروف می باشد و در بسیاری از الگوریتم های دیگر از این روش استفاده های فراوانی می شود. حتی روش هایی که ربط چندانی به نظریه هافمن ندارند و تفاوت های زیادی با هافمن دارند تاثیر این الگوریتم قابل مشاهده است، در این زمینه مقاله ای به نام "Code and Parse Trees for Lossless Source Encoding" وجود دارد که به روش های جدید در این زمینه اشاره می کند. امروزه هافمن علاوه بر رمز نگاری و در داخل الگوریتم های فشرده سازی و نرم افزار های مربوطه، برای ساخت دیتابیس های جدید مانند، H-Base و تعدادی دیگر، و در طبقه بندی اطلاعات دریافتی در سیستم های بزرگ پردازش اطلاعات دنیا، فشرده سازی هافمن از روش های پایه و کاربردی محسوب می شود، و جدای آن بسیاری از روش های دیگر نیز سرچشمه گرفته از این روش هستند به طوری که عموم روش های پرکاربرد و پراستفاده امروزی در زمینه پردازش اطلاعات و بحث های مربوطه، بخش قابل توجهی از خود را مدیون فشرده سازی هافمن و الگوریتم های دیگری که با کمک و بر مبنای هافمن ساخته شده اند، می باشند.

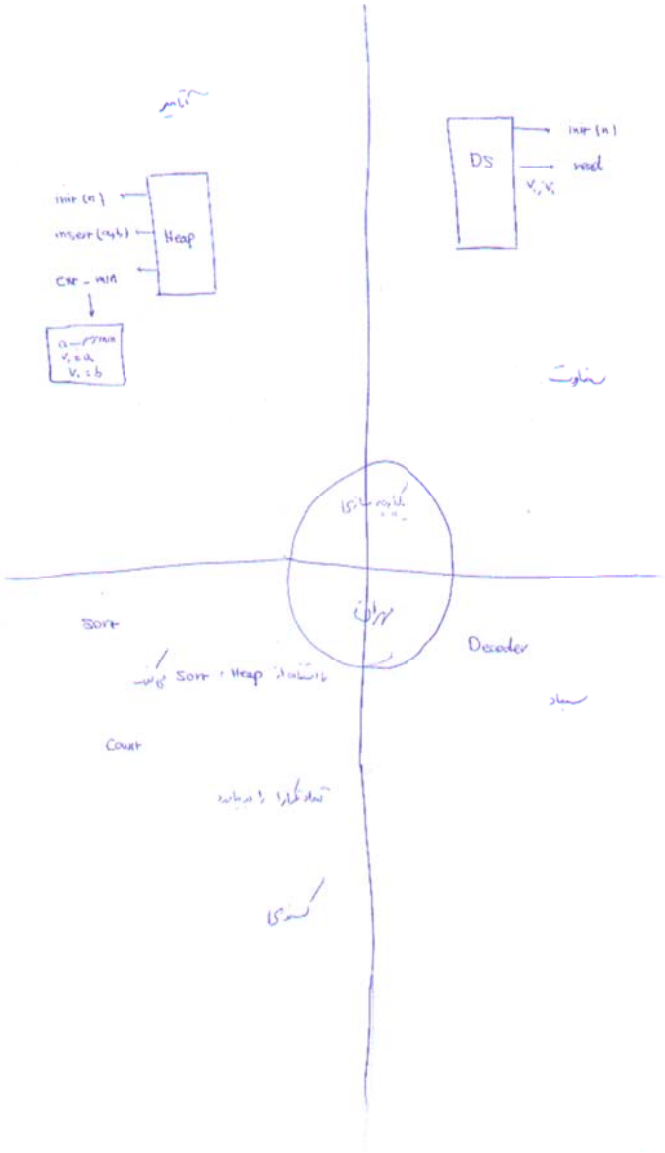
طراحی درخت هافمن به این صورت است که ۱- در ابتدا برای هر کاراکتر، خانه ای در حافظه قرار داده و احتمال حضور آن را مشخص می کنیم، ۲-سپس در هر مرحله، دو خانه درخت، با کمترین احتمال حضور را حذف کرده و به جای آنها خانه جدیدی با احتمال حضور برابر مجموع آن دو قرار می دهیم. و رسیدن به هر کدام از آن دو آن خانه را با بیت های ۰ و ۱ تناظر می دهیم. ۳- سپس در و رمز نگاری متن توسط این روش، به ازای هر کاراکتر، از آن کاراکتر تا بالای درخت را پیمایش می کنیم و بیت

های آمده را با ترتیب برعکس، یعنی از بالا به پایین درخت در bit string در نظر گرفته شده قرار می دهیم. ۴- همچنین برای رمز گشایی و بازخوانی اطلاعات، از ابتدای string شروع می کنیم و بیت ها را به ترتیب می خوانیم، و بر اساس آنها، درخت را پیمایش می کنیم و در هر مرحله که به راس نهایی، یعنی راسی که نماینده یک کاراکتر هست، می رسیم، آن کاراکتر را در متن خروجی لحاظ کرده و ادامه کار را از راس ریشه، پی می گیریم تا به این ترتیب کل متن بازخوانی شود.

از بخش های معروف این الگوریتم، n-Array Huffman encoding می باشد که روشی اپتیمال برای درخت بندی یک متن با n آرایه می باشد. همچنین روش مشابهی نیز برای کدهای به طول توانی از دو وجود دارد. روش دیگری adaptive Huffman coding می باشد که به احتما حضور هر کاراکتر را به صورت داینامیکالی و بر اساس تکرار حضور آن تا این جای ورودی، در نظر می گیرد. همیچنین length limited Huffman coding وجود دارد که به روشی از هافمن اشاره می کند که در آن طول استفاده شد از برای هر کاراکتر باید کمتر از مقدار مشخص شده ای باشد و به همین ترتیب روش Huffman coding with unequal letter cost وجود دارد که در آن طول بیتهای استفاده شده برای هر کاراکتر نباید بیشتر از طول واقعی تعداد بیتهای استفاده شونده توسط آن کاراکتر باشد. همچنین روش the optimal alphabetic binary trees روش بسیار اپتیمال و مناسب برای این الگوریتم، ارائه می دهد که در آن بیت های تخصیص داده شده را به صورت numerical و مرتب شده در نظر می گیرد.

۲- هماهنگی و کار تیمی

۲-۱- جلسه اوّل گروه



در روز پنج شنبه در تاریخ ۱۳۹۱/۴/۱ گروه در ساعت 30:5 یک جلسه برای تعیین وظایف را برگزار کرد. این جلسه به مدت 2 ساعت طول کشید و در طی آن ما با بحث های گروهی به تحلیل مسأله و راه حل آن پرداختیم سپس یک چهارچوب کلی برای نوشتن کد ایجاد کردیم که تصویر آن در زیر آمده است. در طول این بحث ها به هر کسی مسؤلیتی داده شد تا آن مسؤلیت ها را انجام دهند. در این راستا به آقای سجاد جلالی مسؤلیت نوشتن Decoder داده شد و به آقای آقامیر مسؤلیت نوشتن Heap داده شد و به آقای سخاوت مسؤلیت نوشتن درخت هافمن داده شد و به آقای کسنوی مسؤلیت نوشتن خواندن از فایل و محاسبه احتمال ها داده شد. به آقای مهران خلدی نیز مسؤلیت یکپارچه کردن کد ها و نوشتن باینری سرچ و نوشتن کد main و درست کردن کد داده شد.

بعد از جلسه‌ی حضوری اوّل، و تعیین کلّیات طرح، حدوداً تعیین شد که هر کس کدام بخش از پروژه را پیاده‌سازی می‌کند. همچنین برای اینکه کسی منتظر کار بقیه نماند، تلاش کردیم تا برای تمام ماژول‌ها interface مشخصی را جهت صدا شدن توابع تعیین کنیم تا جزئیات پیاده‌سازی بخش‌های مختلف به هم وابسته نباشند.

۲-۲- سامانه github

به همین دلیل، برای ایجاد هماهنگی بیشتر، تصمیم گرفتیم پروژه را با استفاده از یک version control system پیش ببریم تا راحت‌تر در جریانِ پیشرفتِ کار هم قرار بگیریم. از بین گزینه‌های ممکن github را انتخاب کردیم.

آدرس پروژه بر روی گیت‌هاب: https://github.com/SeMeKh/HuffMIPS

به این ترتیب تاریخچه‌ی تمامِ فعالیت‌های اعضاء در history پروژه قابل مشاهده است.

دوست داشتیم با استفاده از ساخت issue برای هر بخش از پروژه و assign کردن آن‌ها به افراد، علاوه بر ایجاد نظم بیشتر، یک مستندسازی ضمنی برای واضح بودن مسیر پیشرفت پروژه داشته باشیم، اما به دلیل وقتِ زیادی که نیاز داشت، این ایده عملی نشد.

امیدواریم بتوانیم بخشی از این مستندات را نیز ترجمه و در ویکی پروژه قرار دهیم تا مستندات در دسترس عموم باشد.

همچنین از تمام پیشنهادات برای بهبود پروژه استقبال می‌کنیم.

۳- ساختار برنامه:

برنامه به دو بخش اصلیِ encoder و decoder تقسیم می‌شد.

الگوریتم کلی encoder:

۱. ورودی خوانده و ذخیره می‌شود

۲. تعداد تکرارهای هر سیمبل محاسبه می‌شود

۲.۱.ابتدا ورودی را مرتب می‌کنیم

۲.۱.۱.داده‌ها را وارد داده‌ساختار heap می‌کنیم

۲.۱.۲.داده‌ها را به ترتیب استخراج می‌کنیم

۲.۲.حال تمام تکرارهای یک سیمبل پشت سر هم قرار خواهند گرفت.

۲.۳.جدولی از سیمبل‌ها به همراه فراوانی محاسبه می‌شود

۳. الگوریتم هافمن را جهت ساخت درخت اجرا می‌کنیم

۳.۱.دو سیمبل که کمترین تعداد تکرار را دارند می‌یابیم و x و y می‌نامیم.

۳.۲.x , y را از مجموعه‌ی سیمبل‌ها حذف می‌کنیم.

۳.۳.سیمبل مجازی V را با نرخِ تکرارِ f(V) = f(x) + f(y) ایجاد و به مجموعه‌ی سیمبل‌ها اضافه می‌کنیم.

۳.۴.در درخت هافمن، V را پدر x و y قرار می‌دهیم.

۳.۵.تا زمانی که لااقل دو سیمبل در مجموعه‌ی سیمبل‌ها وجود دارد به گام اوّل برمی‌گردیم.

۴. هر سیمبل از ورودی با توجه به درخت هافمن به دنباله‌ی صفر و یک تبدیل می‌شود

۵. با چسباندن دنباله‌ی صفر و یک‌ها، می‌توان خروجی را تولید کرد.

۶. برای راحت بودن فرآیند decoding، در فایل خروجی ابتدا درخت و سپس متن encode شده می‌آید.

الگوریتم کلی decoder:

۱. ابتدا ساختار درخت از ورودی خوانده می‌شود.

۲. سپس متن encode شده از ورودی خوانده می‌شود.

۳. درخت هافمن با توجه به دنباله‌ی صفر و یک‌ها که در ورودی آمده پیمایش می‌شود.

۴. با رسیدن به هر رأس برگ در درخت هافمن یک کاراکتر در خروجی چاپ می‌شود.

۳-۱- ماژول های برنامه

نام فایل	وظیفه
encoder/bsearch.asm	پیاده‌سازی الگوریتم جستجوی دودویی جهت تناظر شماره‌ی سیمبل‌های استفاده شده، به اعداد کوچک جهت ذخیره‌سازی در درخت هافمن.
encoder/encoder.asm	نقطه‌ی ورود برنامه‌ی رمزنگار است. پس از خواندن ورودی، آن را پردازش می‌کند، و با کمک huffman_tree و heap داده‌ساختار مورد نیاز درخت هافمن را آماده می‌کند. سپس روال چاپ خروجی را فراخوانی می‌کند.
encoder/freq.asm	ماژولی که موظف است با دریافت یک آرایه‌ی مرتب، سیمبل‌های متفاوتِ آن را به همراه تعداد تکرارهای هر کدام استخراج کند.
encoder/heap.asm	ماژولی که موظف است داده‌ساختار min_heap را جهت استفاده در الگوریتم مرتب‌سازی و در حین ساخت درخت هافمن ارائه کند
encoder/huffman_tree.asm	ماژولی که داده‌ساختار درخت مناسب را جهت ساخت درخت هافمن ارائه می‌کند. همچنین امکان تبدیل سیمبل به دنباله‌ای از صفر و یک را ارائه می‌کند.
encoder/output.asm	ماژولی که با فرض آماده بودن تمامی داده‌ساختارهای مورد نیاز، خروجی را (که در ابتدا به صورت دنباله‌ای از صفر و یک است) به صورت دنباله‌ای از بایت‌ها ذخیره می‌کند.
encoder/sort.asm	ماژولی که در آن با دریافت یک آرایه و اندازه‌ی آن، محتوای آن را می‌کند.
decoder/decoder.asm	خروجی هافمن را می گیرد و آنرا decode نموده و متن اولیه را بازیابی می کند.
merge.bash	فایل های برنامه را ادغام می نماید جهت اسمبل کردن توسط spim
test_all.bash	انجام تست بر روی ورودی و نمایش درستی و Compression Ratio

۳-۲- پیاده سازی Heap

Heap:

این هرم، هرم کمینه است و مورد استفاده درخت هافمن و مرتب سازی قرار می گیرد. این قسمت ۳ تابع دارد که از بیرون به آن

می توان دسترسی پیدا کرد.

۱- heap_init:

در این تابع ما یک هرم کمینه جدید تعریف می کنیم و حافظه را به این هرم اختصاص می دهیم.

در این خطوط:

```
li $v0, 9
sll $a0, $a0, 4
syscall
sw $v0, Heap_Base
```

ما یک حافظه داینامیک برای هرم می گیریم و آدرس پایه آن را در Heap_Base قرار می دهیم.

طول این حافظه توسط کسی که این تابع را صدا می زند تعیین می شود. به این گونه که این مقدار را در $a0 قرار می دهد فقط نکته ای که در این قسمت است این است که این نوع حافظه گرفتن بایت-بایت صورت می پذیرد پس باید به اندازه 4$a0 حافظه بگیریم ولی به خاطر این که بهینه شود ما هم key و هم value را در یک جا ذخیره می کنیم و 8$a0 حافظه می گیریم. برای هر Key ۴ خانه جلوتر آن Value است.

در خطوط:

```
li $v0, 1
sw $v0, Heap_CurrentSize
```

ما اندازه کنونی هرم را به ۱ تغییر می دهیم. این ۱ به خاطر این است که می خواستیم آرایه از ۱ شروع شود و 1-base باشد. با این کار کد بهینه تر می شود.

۲- heap_insert :

در این تابع رجیستر های $t0، $t1، $t2، $t3 و $t4 بدون در نظرگرفتن اینکه ممکن است بقیه از آن ها استفاده کنند استفاده شده است و رجیستر $s0 را قبل از استفاده در stack ذخیره می کند.

در این تابع ورودی در $a0 و $a1 داده می شود که به ترتیب Key و Value هستند و هرم بر مبنای Key کمینه است.

در خط های:

```
lw $s0, Heap_Base
lw $t0, Heap_CurrentSize
sll $t1, $t0, 3
add $t1, $t1, $s0
```

```
sw $a0, 0($t1)
sw $a1, 4($t1)
addi $t0, $t0, 1
sw $t0, Heap_CurrentSize
```

در این خط های کد ما $a0 و $a1 را در حافظه ی گرفته شده ذخیره می کنیم. سپس باید هرم را آپدیت کنیم تا هرم درست شود برای این کار از یک تابع فرعی به نام Heap_InsertUpdate استفاده می کنیم.

۳- heap_extract_min:

در این تابع ما کمینه ی هرم را به عنوان خروجی می دهیم و هرم را به روز می کنیم.

در این تابع رجیستر های $t0، $t1، $t2، $t3 و $t4 بدون در نظرگرفتن اینکه ممکن است بقیه از آن ها استفاده کنند استفاده شده است و رجیستر $s0 را قبل از استفاده در stack ذخیره می کند.

خروجی این تابع $v0 و $v1 است که به ترتیب مربوط به Key و Value کمینه می باشد.

در خط های:

```
lw $s0, Heap_Base
lw $t0, Heap_CurrentSize
addi $t0, $t0, -1
lw $v0, 8($s0)
lw $v1, 12($s0)
sll $t1, $t0, 3
add $t1, $t1, $s0
lw $t2, 0($t1)
sw $t2, 8($s0)
lw $t2, 4($t1)
sw $t2, 12($s0)
sw $t0, Heap_CurrentSize
```

در این خط ها عضو اول با عضو آخر جایگزین می شود و بعد از آن تابع Heap_Update را صدا می زند تا هرم درست شود.

این ۳ تابع بالا تابع های اصلی برنامه هستند و تابع های فرعی را از این به بعد نشان می دهیم.

۱-Heap_Parrent :

این تابع پدر یک ورودی در هرم را بر می گرداند.

ورودی آن $a0 است که شماره اندیس آن عضو است. این اندیس ها 1-base هستند.

خروجی آن $v0 است که اندیس پدر را برمی گرداند.

خط زیر:

```
srl $v0, $a0, 1
```

این کار را می کند که $a0 را بر ۲ تقسیم می کند و در $v0 می ریزد.

۲- Heap_LeftChild:

این تابع ورودی می گیرد و بچه سمت چپ را بر می گرداند.

ورودی در $a0 است و خروجی در $v0 ذخیره می شود.

در خط:

```
sll $v0, $a0, 1
```

$v0 برابر 2$a0 است می شود و اندیس بچه چپ را می دهد.

۳-Heap_RightChild :

این تابع ورودی می گیرد و بچه سمت چپ را بر می گرداند.

ورودی در $a0 است و خروجی در $v0 ذخیره می شود.

در خط های:

```
sll $v0, $a0, 1
add $v0, $v0, 1
```

$v0 برابر 2$a0+1 است که بچه سمت راست را می دهد.

۴-Heap_InsertUpdate:

وقتی که insert را صدا می زنیم همان طور که گفتیم هرم خراب می شود بنابراین برای درست کردن هرم باید این تابع صدا زده

شود در این قسمت ما عددی را که اضافه شده است با پدرش مقایسه می کنیم اگر کوچکتر بود جای آن ها را عوض می کنیم در این قسمت لوپ آن بهینه شده است.

در $a0، اندیس این عدد اضافه شده وجود دارد.

در خطوط:

```
ble $t0, 8, Heap_IUD_End
Heap_IUD_Loop:
        add $t1, $s0, $t0
        srl $a0, $t0, 3
        jal Heap_Parrent
        sll $v0, $v0, 3
        add $t2, $s0, $v0
        lw $t3, 0($t1)
        lw $t4, 0($t2)
        ble $t4, $t3, Heap_IUD_End
        sw $t4, 0($t1)
        sw $t3, 0($t2)
        lw $t3, 4($t1)
        lw $t4, 4($t2)
        sw $t4, 4($t1)
        sw $t3, 4($t2)
        add $t0, $v0, $0
        bgt $t0, 8, Heap_IUD_Loop
Heap_IUD_End:
```

در حال چک کردن و رفتن به پدر هستیم.

۵- Heap_Update:

وقتی که کمینه را حذف می کنیم باید به جای آن کمینه بعدی جایگزین شود که برای این کار از این تابع استفاده می شود در این تابع ما هر بار کوچکترین بچه را انتخاب می کنیم و بعد آن را به جای پدر قرار می دهیم آن قدر این کار را انجام می دهیم که یا در یک مرحله هرم درست شده باشد و پدر از فرزندان کوچکتر شود یا این که دیگر به برگ برسیم.

درخطوط:

```
jal Heap_LeftChild
add $t1, $v0, $0
jal Heap_RightChild
add $t2, $v0, $0
ble $t0, $t1, Heap_UD_secondIf
```

```
        sll $t3, $a0, 3
        sll $t4, $t1, 3
        add $t3, $s0, $t3
        add $t4, $s0, $t4
        lw $t3, 0($t3)
        lw $t4, 0($t4)
        Heap_UD_firstIf:
                ble $t3, $t4, Heap_UD_secondIf
                add $v1, $t1, $0
        Heap_UD_secondIf:
                ble $t0, $t2, Heap_UD_thirdIf
                sll $t3, $v1, 3
                sll $t4, $t2, 3
                add $t3, $s0, $t3
                add $t4, $s0, $t4
                lw $t3, 0($t3)
                lw $t4, 0($t4)
                ble $t3, $t4, Heap_UD_thirdIf
                add $v1, $t2, $0
        Heap_UD_thirdIf:
                beq $v1, $a0, Heap_UD_End
                sll $t1, $a0, 3
                add $t1, $s0, $t1
                sll $t2, $v1, 3
                add $t2, $s0, $t2
                lw $t3, 0($t1)
                lw $t4, 0($t2)
                sw $t4, 0($t1)
                sw $t3, 0($t2)
                lw $t3, 4($t1)
                lw $t4, 4($t2)
                sw $t4, 4($t1)
                sw $t3, 4($t2)
                add $a0, $v1, $0
                jal Heap_Update
        Heap_UD_End:
```

این کار را انجام می دهیم. یعنی چک می کنیم که آیا کوچکترین عضو در بین پدر، بچه راست و بچه چپ را می بینیم و اگر این کوچکترین خود پدر نباشد آن را با پدر عوض می کنیم و به صورت بازگشتی این کار را انجام می دهیم. برای ورودی این تابع اندیس پدر در $a0 قرار می گیرد.

توضیحات نویسنده:

در ابتدا در جلسه ای که در روز پنجشنبه مورخ ۹۰/۴/۱ تشکیل شد شرکت کرده و گروه تصمیم گرفت که قسمت Heap را به من واگذار کند.

قبل از این که من این پروژه را شروع کنم. ابتدا مطالعاتی در رابطه با چگونگی گرفتن حافظه داینامیک انجام دادم. من با رفتن به سایت http://courses.missouristate.edu/KenVollmar/MARS/Help/SyscallHelp.html فهمیدم که sbrk برای گرفتن حافظه داینامیک است و باید $v0 را برابر ۹ قرار داده و تعداد خانه های مورد نیاز را در $a0 قرار دهیم. که این با انجام این دستور آدرس پایه حافظه گرفته شده در $v0 قرار داده می شود.

با این مطالعه پروژه خود را شروع کرده و کد اولیه ای برای Heap زدم که دارای همین تابع های فعلی بود. بعد از یک مدت معلوم شد که این کد یک Bug دارد این موضوع بوسیله آقای خلدی به من اطلاع داده شد که با Debug کردن متوجه شدم که در این خط های کد که باید swap انجام می دادم این کار را نمی کردم:

```
lw $t3, 0($t1)
lw $t4, 0($t2)
sw $t4, 0($t1)
sw $t3, 0($t2)
```

و به اشتباه این گونه نوشته بودم

```
lw $t3, 4($t1)
lw $t4, 4($t2)
sw $t4, 4($t2)
sw $t3, 4($t1)
```

در ابتدا من این کد را با 2 پایه آدرس یکی برای Value و دیگری برای Key نوشته بودم ولی به پیشنهاد آقای خلدی مبنی بر بهینه کردن آن من این ۲ پایه را به یک پایه تبدیل کردم با این تبدیل دوباره Bug در این کد بوجود آمد که بدین صورت بود:

در Heap_InsertUpdate: در خط ble $t0, 8, Heap_IUD_End به اشتباه به جای ۸، ۳۲ گذاشته بودم.

با درست کردن این مشکلات و بهینه کردن Loop در داخل Heap_InsertUpdate کار خود را در تکمیل این قسمت به پایان رساندم.

۳-۳- پیاده سازی Output و Decoder

Output:

خروجی دو قسمت کاملا مجزا دارد:

الف) قسمت اطلاعاتی که برای decode کردن مورد نیاز است :

ابتدا تعداد سیمبل های مختلف چاپ شده است

سپس nسیمبل که مرتب شده اند.

در خط بعدی فرزندان سمت چپ هر گره در درخت هافمن چاپ شده است.

در آخرین سطر از این قسمت هم فرزندان سمت راست هر گره در درخت هافمن چاپ شده است.

با استفاده از این اطلاعات میتوان به راحتی با حرکت کردن روی درخت به این صورت که با دیدن بیت صفر به فرزند چپ برود ، و با دیدن بیت یک به سمت راست ، به سیمبل مورد نظر دست یافت.

ب) در این قسمت ابتدا تعداد بیت های جواب چاپ میشود

و سپس متن کدشده به صورت بایت بایت (یک بایت مانند بافر عمل می کند) چاپ می شود.

Decoder:

با ساختن درخت و خواندن ورودی کد شده ، آن را ذخیره میکند و از آخر به اول ورودی را پیمایش کرده و همزمان روی درخت از ریشه حرکت میکند. هنگامی که به یک سیمبل رسید آن را چاپ میکند و دوباره به ابتدای ریشه باز می گردد.

در ورودی خواندن ، ما متن کد شده را در بایت ریخته ایم و اینجا دوباره برای خواندن اعداد از یک اشاره گر به بیت مورد نظر بایت استفاده میکنیم که در هر مرحله آن را به روز رسانی میکنیم. (به راست شیفت می دهیم) با استفاده از این روش به راحتی خروجی ایجاد می شود.

باید توجه داشته باشیم که ورودی فایل decoder به گونه ای تنظیم شده است که اگر از آخر به اول پردازش شود ورودی اصلی encoder تولید شود نه به ترتیب برعکس.

۳-۴- پیاده سازی Huffman_Tree

پیاده سازی درخت هافمن شامل سه رویه ی init و merge و encode می باشد. همچنین آرایه های لازم برای نگه داری اندیس فرزند راست و چپ هر رأس باید global باشد تا در بخش output چاپ شود.

رویه ی Init:

این رویه ورودی n را می گیرد و حافظه لازم برای حداکثر 2*n رأس درخت را تخصیص می دهد و مقادیر اولیه را برای آرایه های فرزند چپ و فرزند راست و پدر قرار می دهد (با توجه به مثبت بودن اندیس رئوس، برای Null از 1- مقدار استفاده شد)

رویه ی Merge:

این رویه شماره دو رأس از جنگل هافمن فعلی را می گیرد و با اضافه کردن یک رأس جدید و قرار دادن دو رأس ورودی در فرزند های چپ و راست رأس جدید، شماره رأس جدید را بر می گرداند.

رویه ی Encode:

این رویه یک آرایه از شماره برگ های درخت را گرفته و مسیر هر برگ آرایه تا ریشه را پیموده و رشته ی encode شده ی مربوط به آن برگ را به خروجی اضافه می کند.

این رویه تعداد بیت های رشته ی encode شده و آدرس آنرا بر می گرداند.

لازم به ذکر است با توجه به آنکه باید از برگ به ریشه حرکت کرد، این آرایه رشته ی encode شده را به صورت معکوس تولید می کند و برای decode کردن آن لازم است رشته به صورت معکوس خوانده شود.

۴- اجرای برنامه در محیط های مختلف

۴-۱- MARS

از آنجا که اکثر کد در محیط همین نرم‌افزار توسعه داده شده، اجرای آن نیز به راحتی ممکن است. تنها کافیست تا تمامِ فایل‌های مربوطه در یک پوشه قرار بگیرند.

Mars قابلیت اجرا در Console را نیز دارا می باشد که توسط دستور زیر محقق می شود:

```
java -jar mars.jar <files> -nc
```

۴-۲- SPIM

علی‌القاعده شبیه‌ساز MARS باید با SPIM کاملاً سازگار باشد. اما در عمل، وقتی می‌خواستیم برنامه را در این شبیه‌ساز اجرا کنیم به مشکلاتی بر خوردیم که برای رفع آن‌ها، مراحل زیر طی شد:

- از یک اسکریپت ساده (merge.bash) برای ادغام تمام فایل‌ها به صورتِ تک‌فایل all.asm استفاده کردیم.

  توضیح: برای بهبود ساختار کد و هماهنگی راحت‌تر در هنگام استفاده از github، تلاش کردیم تا بخش‌های مختلف در ماژول‌های مجزّا و بالطّبع در فایل‌های جداگانه قرار بگیرند. اما وقتی خواستیم برنامه را با SPIM اجرا کنیم، فهمیدیم که اگرچه امکان بارگذاری چند فایل به صورت تعاملی وجود دارد، اما از طریق خط فرمان این کار امکان‌پذیر نیست. در واقع، برای اجرای یک فایل توسط SPIM از دستور spim -file main.asm استفاده می‌کردیم، اما با اضافه کردن نام بقیه‌ی فایل‌ها، به جای اینکه تمام فایل‌ها بارگذاری شوند، تنها فایل اوّل بارگذاری می‌شد و بقیه‌ی فرمان به عنوان آرگومانِ اجرایی برنامه درنظر گرفته می‌شد. به همین دلیل مجبور بودیم تمام فایل‌ها را در یک فایل نهایی تلفیق کنیم.
- حذف دستورات subi و جایگزین کردن آن‌ها با معادل addiشان

  جالب بود که SPIM از شبه‌دستور subi پشتیبانی نمی‌کرد (در حالی که MARS آن را اجرا می‌کرد) به همین دلیل کاربردهای subi در کد با دستور معادل addi جایگزین شدند.
- در نهایت توانستیم کد را به نحوی تغییر دهیم که در محیط شبیه‌ساز SPIM نیز قابل اجرا باشد.

۴-۳- QEMU:

می‌دانیم که MARS و SPIM هر دو شبیه‌ساز[۱] محیط اجرای MIPS هستند. به این معنی که دستورات اسمبلی را مستقیماً اجرا نمی‌کنند، بلکه آن‌ها را «تفسیر» و اجرا می‌کنند. در حالی که یک مقلّد[۲] یک گام به واقعیت نزدیک‌تر است. در emulator، عملاً کارایی یک پردازنده‌ی دلخواه (در این مورد MIPS) تقلید می‌شود و دستورات ماشین به آن پردازنده‌ی مجازی جهت پردازش انتقال می‌یابند. به این ترتیب محیط شبیه‌سازی شده توسط این گونه نرم‌افزارها به محیط واقعی نزدیک‌تر بوده و معمولاً از سرعت بهتری نیز برخوردارند.

- برایمان جالب بود که بتوانیم کد خودمان را بیرون از محیط شبیه‌ساز و در یک ماشین مجازی بسنجیم تا مطمئن شویم که زبان اسمبلی و پردازنده‌ی میپس یک توهّم نیست! اما...
- ابتدا نرم‌افزار qemu که یک نرم‌افزار قدرت‌مند جهت راه‌اندازی ماشین‌های مجازی با ساختارهای مختلف پردازنده است را نصب می‌کنیم. این کار در توزیع‌های مبتنی بر دبیان[۳] لینوکس از طریق دستور زیر امکان‌پذیر است:

```
sudo apt-get install qemu-system
```

- گام دوّم نصب و راه‌اندازی یک سیستم عامل بر روی ماشین مجازی خواهد بود، تا کارها را تسهیل کند. خوشبختانه دبیان برای پردازنده‌ی میپس نیز پورت شده. راهنمای نصب و راه‌اندازی دبیان بر روی qemu نیز از اینجا قابل دست‌یابیست.
- پس از نصب و راه‌اندازی سیستم عامل، باید ابزار مورد نیاز کامپایل و اسمبل را نصب کنیم. در توزیع‌های مبتنی بر دبیان:

```
sudo apt-get update && sudo apt-get install build-essential
```

- برای تولید کد اسمبلی از کد سی کافیست از دستور زیر استفاده کنیم.

```
g++ huffman.cpp -S
```

- با مقایسه‌ی huffman.s (که توسط کامپایلر تولید شده) و فایل اسمبلی که توسط خودمان تولید شده می‌فهمیم که حجم برنامه به شدّت کمتر شده!
- بعد از اینکه توانستیم با یک سیستم عامل کامل روی ساختار میپس کد کامپایل و اجرا کنیم، تصمیم گرفتیم تا یک benchmark داشته باشیم و سرعت کد خودمان و کد تولیدشده توسط کامپایلر را مقایسه کنیم:

---

۱ Simulator
۲ Emulator
۳ Debian

- کد اسمبلی خودمان را با استفاده از کامپایلر:

```
g++ all.asm
```

- البته کامپایل موفقیت‌آمیز بود، اما برنامه ورودی نمی‌خواند و خروجی نمی‌نوشت!
- بعد از بررسی فهمیدیم که syscallها در این محیط با MARS و SPIM فرق دارند و چون نتوانستیم یک manual مناسب برای پیدا کردن شماره‌ی syscallهای مناسب پیدا کنیم، مجبور شدیم از اجرای برنامه در qemu صرف‌نظر کنیم. :(

۵- پیشنهادهایی برای ادامه‌ی پروژه

- **پیاده‌سازی DEFLATE**

deflate یک الگوریتم فشرده‌سازی است که از کدینگ هافمن برای کم کردن حجم اطلاعات انتقالی استفاده می‌کند. تغییر کد این پروژه به نحوی که با این الگوریتم سازگار باشد کار جالبی به نظر می‌رسد.

- **تغییر کد به نحوی که روی یک پردازنده‌ی MIPS واقعی قابل اجرا باشد**

در طول بررسی‌ها دیدیم که عملاً محیط شبیه‌سازهای ما با پردازنده‌های واقعی تفاوت‌هایی دارد. بررسی این تفاوت‌ها و اجرای واقعی این پروژه (و در کل، هر پروژه‌ی دیگری) حتماً آموزنده خواهد بود.

۶-منابع

- http://en.wikipedia.org/wiki/Heap_%28data_structure%29
- http://en.wikipedia.org/wiki/Huffman_coding
- http://www.aurel32.net/info/debian_mips_qemu