Le guide du débutant à Redcode

Version 1.22

Contenu

Préface

Il n'y a pas trop de débutants intéressés par le jeu de la guerre de base de ces jours. Bien sûr, cela est tout à fait naturel - pas que beaucoup de gens considèrent l'optimisation du code d'assemblage pour être amusant de toute façon - mais l'une des raisons pour le seuil de démarrage élevé peut être la difficulté de trouver des informations sur les bases mêmes de la partie. Certes, il ya beaucoup de bons documents autour, mais la plupart d'entre eux sont soit trop technique, pas à jour, trop difficiles à trouver ou tout simplement incomplète.

C'est pourquoi j'ai décidé d'écrire ce guide. Mon but est de guider les nouveaux arrivants dès leur premier contact avec la guerre de base et Redcode au point où ils peuvent écrire un travail (en cas d'échec) guerrier, et sont en mesure de procéder à la substance plus technique.

Pour être honnête, je suis encore un débutant dans ce jeu moi-même. Je sais que la langue assez bien, mais n'ont pas encore produit un guerrier vraiment réussi. Mais j'ai décidé de ne pas attendre jusqu'à ce que je suis devenu plus expérimenté et au lieu d'écrire ce guide dès que possible pendant que j'ai encore une nouvelle mémoire de ce que c'est que d'être un nouveau joueur du mal à comprendre les particularités de la partie.

Ce guide est destiné aux *mêmes* débutants. Aucune connaissance préalable de tout langage d'assemblage (ou la programmation en général) devrait être nécessaire, tout en sachant l'idée générale devrait aider à comprendre les termes de base. Redcode, en particulier les versions modernes, peut ressembler à un code d'assemblage, mais il est plus abstrait que la plupart et très différent dans les détails de tout autre langage d'assemblage.

La saveur de Redcode utilisé dans ce guide est (surtout) le courant *de facto* standard, la ICWS 94 Projet de norme avec pMARS 0,8 extensions. (Un peu comme les extensions Netscape au format HTML ... Hmm ... Heureusement, nous n'avons toujours pas un Microsoft

Corewar Simulator. Peut-être qu'ils pensent trop petit du marché.) La norme 88 plus tôt sera mentionné brièvement, mais ce guide est la plupart du temps sur la norme 94. Pour ceux qui veulent l'apprendre, il ya beaucoup de '88 tutoriels disponibles sur le Web.

Important : Il n'existe aucun moyen simple d'enseigner Redcode - ou n'importe quel langage de programmation - de façon strictement linéaire. Bien que j'ai essayé d'organiser ce guide dans un ordre quelque peu sensible, *si vous voulez sauter autour, par tous les moyens le faire* . C'est ce que le contenu de l'article est pour.

Pour maintenir une cohérence à tous, j'ai souvent été obligé de vous montrer les choses et ensuite expliquer les quelques chapitres plus loin. Si vous n'avez pas l'air de comprendre quelque chose, à lire pendant un moment. Si vous n'arrivez toujours pas à comprendre, essayer de parcourir pour voir si il est expliqué dans un autre chapitre.

Tout le monde apprend de façon différente, et si l'ordre que vous décidez de lire les chapitres en est probablement meilleur que celui que j'ai choisi. Mais si vous pensez que quelque chose est ennuyeux et laisser complètement non lus, les chances sont que vous allez manquer quelque morceau important si l'information. J'ai essayé de marquer des parties importantes avec *l'accent* , si vous savez où s'arrêter et de réfléchir, mais j'essaie de tout lire attentivement. Je ne peux tout simplement pas tout épeler, ou ce guide deviendrais trop long à lire.

Introduction à la Guerre du Coeur

Qu'est-ce que la guerre Core?

Guerre du Cœur (ou *de base Wars*) est un jeu de programmation où les programmes de montage essaient de détruire l'autre dans la mémoire d'un ordinateur simulé. Les programmes (ou *guerriers*) sont écrits dans un langage spécial appelé *Redcode* , et gérés par un programme appelé *MARS* (*Mémoire tableau Redcode Simulator*).

Les deux Redcode et l'environnement MARS sont très simplifiées et abstraites par rapport aux systèmes informatiques ordinaires. C'est une bonne chose, puisque les programmes d'armes chimiques ont été écrites pour la performance, pas pour plus de clarté. Si le jeu utilise un langage d'assemblage ordinaire, il pourrait y avoir deux ou trois personnes au monde capables d'écrire un guerrier efficace et durable, et même ils seraient probablement pas en mesure de comprendre pleinement. Il serait certainement difficile et plein de potentiel, mais il serait probablement prendre des années pour atteindre même un niveau modéré de compétence.

Comment ça marche?

Le système dans lequel les programmes sont exécutés est assez simple. Le *noyau* (la mémoire de l'ordinateur simulé) est un réseau continu d'instructions, vide, sauf pour les programmes concurrents. Le noyau s'enroule autour, de sorte que, après la dernière instruction vient à nouveau à la première.

En fait, les programmes n'ont aucun moyen de savoir où se termine le noyau, car il n'y a pas des adresses absolues. Autrement dit, l'adresse 0 ne signifie pas que la première instruction dans la mémoire, mais l'instruction qui contient l'adresse. L'instruction suivante est 1, et le précédent évidemment -1.

Comme vous pouvez le voir, l'unité de base de la mémoire dans la guerre Core est une instruction, au lieu d'un seul octet comme d'habitude. Chaque instruction Redcode contient trois parties: la *OpCode* lui-même, l'adresse de source (aka l' *A-champ*) et l'adresse de destination (le *-champ B*). Alors qu'il est possible par exemple de déplacer des données entre l'A-champ et le champ B, en général vous avez besoin pour traiter les instructions que des blocs indivisibles.

L'exécution des programmes est tout aussi simple. Le MARS exécute une instruction à la fois, et procède ensuite à la suivante dans la mémoire, à moins que l'instruction, il dit explicitement à passer à une autre adresse. S'il ya plus d'un fonctionnement du programme, (comme d'habitude) les programmes exécutent alternativement, une instruction à la fois. L'exécution de chaque instruction prend le même temps, un cycle, que ce soit `MOV` , `DIV` ou même `DAT` (qui tue le processus).

À partir de Redcode

Le jeu d'instructions de Redcode

Le nombre d'instructions dans Redcode a augmenté avec chaque nouvelle norme, à partir du nombre initial de environ 5 pour le courant 18 ou 19. Et cela ne comprend même pas les nouveaux modificateurs et modes d'adressage qui permettent littéralement des centaines de combinaisons. Heureusement, nous n'avons pas besoin d'apprendre toutes les combinaisons. Il suffit de se rappeler les instructions, et comment les modificateurs de les changer.

Voici une liste de toutes les instructions utilisées dans Redcode:

- `DAT` - données (tue le processus)
- `MOV` - déplacer (copier les données d'une adresse à l'autre)
- `ADD` - ajouter (ajoute un numéro à un autre)
- `SOUS` - soustraire (soustrait un nombre d'un autre)
- `MUL` - se multiplient (multiplie un nombre par un autre)
- `DIV` - fracture (divise un nombre par un autre)
- `MOD` - module (divise un nombre par un autre et donne le reste)
- `JMP` - saut (continue l'exécution d'une autre adresse)
- `JMZ` - sauter si zéro (teste plusieurs sauts et à une adresse si c'est 0)
- `JMN` - sauter si pas nul (teste plusieurs sauts et si elle n'est pas 0)
- `DJN` - décrément et saut si pas nul (décrémente un nombre par un, et les sauts à moins que le résultat est 0)
- `SPL` - split (commence un deuxième processus à une autre adresse)
- `CMP` - comparer (comme `SEQ`)
- `SEQ` - sauter en cas d'égalité (compare deux instructions, et saute l'instruction suivante si elles sont égales)
- `END` - sauter si pas égal (compare deux instructions, et saute l'instruction suivante si elles ne sont pas égales)
- `SLT` - sauter si inférieur (compare deux valeurs, et saute l'instruction suivante si la *première* est inférieure à la *seconde*)
- `LDP` - charge de p-espace (charge un certain nombre de l'espace de stockage privé)
- `STP` - enregistrez-p-espace (enregistre un certain nombre d'espace de stockage privé)
- `NOP` - aucune opération (ne fait rien)

Ne vous inquiétez pas si certains d'entre eux semblent, pour le moins, étrange. Comme je le disais, Redcode est un peu différente de langages d'assemblage plus ordinaires, qui résulte de son caractère abstrait.

L'IMP

La vérité est, les parties les plus importantes de Redcode sont les plus faciles. La plupart des types de guerriers de base ont été inventés avant les nouvelles instructions et modes existaient. Le plus simple, et probablement le premier programme guerre Core est le *diablotin* , publié par AK Dewdney en 1984 l'article original Scientific American que la première Guerre du Coeur présenté au public.

```
        MOV 0, 1
```

Oui, c'est ça. Juste un mauvais `MOV` . Mais que faut-il *faire* ? `MOV` bien sûr des copies d'une instruction. Vous devriez rappeler que toutes les adresses de guerre de base sont liés à l'instruction en cours, de sorte que le diablotin en fait se copie dans l'instruction juste après lui-même.

```
        MOV 0, 1          , ce qui a été exécuté juste
        MOV 0, 1          ; cette instruction sera exécuté suivant
```

Maintenant, l'Imp va exécuter l'instruction qu'il vient d'écrire! Comme il est exactement le même que le premier, il sera une fois de plus se copier une instruction avant, exécuter la copie, et de continuer à aller de l'avant tout en remplissant le noyau avec `MOV` s. Depuis le noyau n'a pas de fin réelle, l'Imp, après avoir rempli l'ensemble de base, atteint sa position de départ à nouveau et continue à tourner joyeusement dans les cercles *à l'infini* .

Donc le Imp crée en fait son propre code tel qu'il l'exécute! Dans la guerre de base, l'auto-modification est une règle plutôt que l'exception. Vous devez être efficace pour réussir, et cela signifie presque toujours changer votre code à la volée. Heureusement, l'environnement abstrait fait un beaucoup plus facile à suivre que dans l'assemblage ordinaire.

BTW, il devrait être évident qu'il n'y a pas de caches en guerre Core. Eh bien, en fait le *courant* instruction est mise en cache de sorte que vous ne pouvez pas le modifier *dans le milieu* de l'exécuter, mais peut-être que nous devrions laisser tout cela pour le plus tard ...

Le Nain

L'IMP a un petit inconvénient, comme un guerrier. Il ne gagnera pas trop de jeux, depuis quand il remplace un autre guerrier, il commence aussi à exécuter le `MOV 0, 1` et devient un lutin, aboutissant à une cravate. Pour tuer un programme, vous auriez à copier un `DAT` sur son code.

C'est exactement ce que l'autre guerrier classique par Dewdney, le *nain* , le fait. Il «bombes» de la base à des emplacements espacés régulièrement avec `DAT` s, tout en s'assurant que ce ne sera pas lui-même frappé.

```
        AJOUTER  # 4, 3         ; exécution commence ici
        MOV 2, @ 2
         JMP -2
         DAT  # 0, # 0
```

En fait, ce n'est pas précisément ce qu'a écrit Dewdney, mais il fonctionne exactement de la même façon. L'exécution commence de nouveau à la première instruction. Cette fois, c'est un `ADD` . L' `ADD` instruction ajoute la source et la destination ensemble, et met le résultat dans la destination. Si vous êtes familier avec d'autres langages d'assemblage, vous pouvez reconnaître le `#` signe comme un moyen de marquage adressage immédiat. Autrement dit, le `ADD` ajoute le numéro 4 de l'instruction à l'adresse 3, au lieu d'ajouter l'instruction de l'instruction 4 3. Depuis le 3 instruction après l' `ADD` est le `DAT` , le résultat sera:

```
        AJOUTER  # 4 , 3
         MOV 2, @ 2          ; prochaine instruction
        JMP -2
         DAT  # 0, # 4
```

Si vous ajoutez deux instructions ensemble, à la fois le A-et B-champs seront additionnés indépendamment les uns des autres. Si vous ajoutez un numéro unique à une instruction, il sera par défaut ajouté au champ de B. Il est tout à fait possible d'utiliser un `#` dans le champ B de l' `ADD` trop. Ensuite, le champ A-serait ajoutée à la B-domaine de la `TDA` lui-même.

Le mode d'adressage immédiat peut sembler simple et familier, mais les nouveaux modificateurs de la norme ICWS 94 lui donnera une toute nouvelle tournure . Mais regardons le Nain premier.

Le `MOV` nous présente une fois de plus avec un autre mode d'adressage: le `@` ou le mode d'adressage indirect. Cela signifie que le `DAT` ne sera pas copié sur lui-même comme il semble (à quoi bon ce que ce serait?), mais sur l'instruction de ses points B-terrain à, comme ceci:

```
        AJOUTER  # 4, 3
         MOV 2, @ 2   .; -
        JMP -2       ; | 2
        DAT  # 0, # 4 ; <- '-. Le B-champ des points MOV ici.
        ...                  |
        ...                  | 4
        ...                  |
        DAT  # 0, # 4 ; <------ "Le champ B de la DAT souligne ici.
```

Comme vous pouvez le voir, la `DAT` sera copié 4 instructions à venir de celui-ci. La prochaine instruction, `JMP` , est tout simplement le processus sauter deux instructions en arrière, vers le `ADD` . Depuis le `JMP` ignore son champ de B je l'ai laissé vide. Le MARS initialise pour moi comme un 0.

Par ailleurs, comme vous le voyez le MARS ne sera pas commencer à tracer de nouvelles chaînes d'adresses indirectes. Si les points d'opérandes indirects à une instruction avec un B-champ, disons, 4, la destination réelle seront 4 instructions après, quel que soit le mode d'adressage.

Maintenant, le `TDA` et le `MOV` seront exécutés à nouveau. Lorsque l'exécution atteint la `JMP` nouveau, le noyau se présente comme suit:

```
        AJOUTER  # 4, 3
         MOV 2, @ 2
         JMP -2            ; instruction suivante
        DAT  # 0, # 8
         ...
        ...
        ...
        DAT  # 0, # 4
         ...
        ...
        ...
        DAT  # 0, # 8
```

Le nain va continuer à tomber `DAT` toutes les 4 instructions, jusqu'à ce qu'il a enroulée autour de l'ensemble du noyau et se soit de nouveau atteinte:

```
        ...
        DAT  # 0, # -8
         ...
        ...
        ...
        DAT  # 0, # -4
         ADD  # 4, 3          ; prochaine instruction
        MOV 2, @ 2
         JMP -2
         DAT  # 0, # -4
         . ..
        ...
        ...
```

```
        DAT  # 0, # 4
         ...
```

Maintenant, l' `ADD` deviendra le `DAT` retour en `# 0, # 0` , le `MOV` effectuera un exercice de futilité en copiant le `DAT` droit où il est déjà, et tout le processus recommence depuis le début.

Cela va naturellement fonctionne pas à moins que la taille du noyau est divisible par 4, car sinon le nain aurait frappé une instruction de 1 à 3 instructions derrière le `DAT` , se tuant ainsi. Heureusement, la taille la plus populaire de base est actuellement de 8000, suivie par 8192, 55400, 800, tous divisibles par 4, de sorte que notre nain doit être sûr

Comme une note de côté, y compris le `DAT # 0, # 0` dans le guerrier ne serait pas vraiment été nécessaire; L'instruction le noyau est initialement rempli, que j'ai écrit que trois points (`...`) est en fait `DAT 0 , 0` . Je vais continuer à utiliser les points pour décrire noyau vide, car il est plus court et plus facile à lire.

Les modes d'adressage

Dans les premières versions de guerre de base des modes d'adressage ne sont immédiats (`#`), la contribution directe (`$` ou rien du tout) et les indirects (B-terrain `@`) modes d'adressage. Par la suite, le mode d'adressage prédécrémentation, ou `<` , a été ajouté. C'est le même que le mode indirect, sauf que le pointeur est décrémenté d'une unité avant que l'adresse cible est calculée.

```
        DAT  # 0, # 5
         MOV 0 < -1         ; instruction suivante
```

Lorsque cette `MOV` est exécutée, le résultat sera le suivant:

```
        DAT  # 0, # 4  ; ---.
        MOV 0 < -1 ; |
        ...          ; | 4
        ...          ; |
        MOV 0 < -1 ; <--- '
```

Le ICWS '94 projet de norme a ajouté quatre modes d'adressage plus, la plupart du temps pour faire face à un champ indirect, pour donner un total de 8 modes:

- `#` - immédiat
- `$` - directe (le `$` peut être omis)
- `*` - Un champ indirect
- `@` - B-champ indirect
- `{` - Un champ indirect avec prédécrémentation
- `<` - B-champ indirect avec prédécrémentation
- `}` - Un champ indirect avec post-incrémentation
- `>` - B-champ indirect avec post-incrémentation

Les modes de post-incrémentation sont similaires à la prédécrémentation, mais le pointeur seront *augmentés* par un *après* l'instruction a été exécutée, comme vous l'aurez deviné.

```
        DAT  # 5, # -10
         MOV -1, } -1       ; instruction suivante
```

sera après l'exécution ressembler à ceci:

```
        DAT  # 6 , # -10 , -.
        MOV -1, } -1   ; |
        ...            ; |
        ...            ; | 5
         ...           ; |
        DAT  # 5 , # -10 ; <- '
```

Une chose importante à retenir sur les modes de prédécrémentation et post-incrémentation est que les pointeurs seront in-/decremented même si elles ne sont pas utilisées pour rien. Donc `JMP -1, < 100` serait diminuer l'instruction 100, même si la valeur désignée n'est pas utilisé pour quoi que ce soit. Même `DAT < 50, < 60` décrémentera les adresses en plus de tuer le processus.

La file d'attente de processus

Si vous regardez le tableau d'instruction quelques chapitres ci-dessus attentivement, vous avez peut-être interrogé sur une instruction

nommée `SPL` . Il n'y a certainement rien de tel dans n'importe quel langage de assemblée ordinaire ...

Très tôt dans l'histoire de la guerre de base, il a été suggéré que l'ajout de multitâche pour le jeu, il serait beaucoup plus intéressant. Depuis les techniques de découpage de temps bruts utilisés dans les systèmes ordinaires ne rentrait pas dans le contexte de la guerre de base abstraite (le plus important, vous aurez besoin d'un OS pour les contrôler), un système a été inventé de sorte que chaque processus est exécuté pour un cycle à son tour .

L'instruction utilisée pour créer de nouveaux processus est le `SPL` . Il faut une adresse en tant que paramètre dans son A-champ, tout comme `JMP` . La différence entre `JMP` et `SPL` est que, en plus de lancer l'exécution à la nouvelle adresse, `SPL` *aussi* continue l'exécution à l'instruction suivante.

Les deux - ou plusieurs - processus ainsi créés se partageront le temps de traitement aussi. Au lieu d'un compteur de processus unique qui montrerait l'instruction en cours, la MARS a une *file d'attente de processus* , une liste des processus qui sont exécutées à plusieurs reprises dans l'ordre dans lequel ils ont commencé. Nouveaux processus créés par `SPL` sont ajoutés juste après le processus en cours, tandis que ceux qui exécutent un `DAT` sera retiré de la file d'attente. Si tous les processus meurent, le guerrier perdra.

Il est important de se rappeler que chaque programme a son *propre* file d'attente de processus. Avec plus d'un programme dans le coeur, ils ne sont exécutés alternativement, *un cycle à la fois quel que soit la longueur de leur file d'attente de processus* , de sorte que le temps de traitement va toujours être divisé en parts égales. Si le programme A a 3 processus, et le programme B seulement 1, l'ordre d'exécution va ressembler:

1. Un programme, processus 1 ,
2. programme B, processus 1 ,
3. Un programme, processus 2 ,
4. programme B, processus 1 ,
5. Un programme, processus 3 ,
6. programme B, processus 1 ,
7. Un programme, processus 1 ,
8. programme B, processus 1 ,

   ...

Et enfin, un petit exemple de l'utilisation de `SPL` . Plus d'informations seront disponibles dans les chapitres suivants.

```
        SPL 0               ; exécution commence ici
        MOV 0, 1
```

Depuis les `SPL` pointe vers lui-même, après un cycle les processus seront comme ceci:

```
        SPL 0               ; deuxième processus est ici
        MOV 0, 1            , premier procédé est ici
```

Après les deux processus ont exécuté, le noyau va maintenant ressembler à:

```
        SPL 0               ; troisième processus est ici
        MOV 0, 1            ; processus de seconde est ici
        MOV 0, 1            , premier procédé est ici
```

Donc, ce code lance évidemment une série de lutins, l'un après l'autre. Il va continuer à le faire jusqu'à ce que les lutins ont encerclé tout le noyau et écraser le `SPL` .

La taille de la file d'attente de processus pour chaque programme est limitée. Si le nombre maximal de processus a été atteint, `SPL` continue exécution *seulement à l'instruction suivante* , la duplication efficacement le comportement de `NOP` . Dans la plupart des cas, la limite du procédé est assez élevé, souvent la même que la longueur de l'âme, mais elle peut être plus faible (voire 1, auquel cas le fractionnement est effectivement désactivé).

Oh, et comme pour thruth étant souvent plus étrange que la fiction, je suis récemment tombé sur une page Web intitulée "opcodes qui aurait dû être". Parmi ceux qui sont vraiment absurdes certains j'ai trouvé " `BBW` - Direction Both Ways ". Comme tous les opcodes étaient censés être fictif, je ne peux que conclure que l'auteur n'était pas familier avec Redcode ..

Les modificateurs d'instruction

La plus importante chose de nouveau apporté par la norme ICWS 94 ne sont pas les nouvelles instructions ou les nouveaux modes d'adressage, mais les modificateurs. Dans l'ancienne norme 88 les seuls modes d'adressage décider quelles parties des instructions sont affectés par une opération. Par exemple, `MOV 1, 2` se déplace toujours une instruction ensemble, tandis que `MOV # 1, 2` se déplace

d'un seul numéro. (Et *toujours* sur le champ de B!)

Naturellement, cela pourrait causer des difficultés. Que faire si vous voulez déplacer que les A-et B-champs d'une instruction, mais pas l'opcode? (Vous aurez besoin d'utiliser `ADD`) Ou si vous voulez passer quelque chose dans le champ de B à la A-champ? (Possible, mais très difficile) Pour clarifier la situation, les modificateurs d'instruction ont été inventés.

Les modificateurs sont des suffixes qui sont ajoutées à l'instruction pour indiquer quelles parties de la source et la destination, il aura une incidence. Par exemple, `MOV . AB 4, 5` serait déplacer le A-champ de l'instruction 4 B dans le champ de l'instruction 5. Il ya 7 modificateurs différents disponibles:

- `MOV A.` - A déplace le champ de la source dans le champ A-de la destination
- `MOV B.` - déplace le champ B de la source dans le champ B de la destination
- `MOV . AB` - déplace l'A-champ de la source dans le champ B de la destination
- `MOV BA.` - déplace le champ B de la source dans le champ A de la destination
- `MOV F.` - déplace les deux champs de la source dans les mêmes champs de la destination
- `MOV X.` - déplace les deux champs de la source dans les *opposés* champs de la destination
- `MOV . j'ai` - déplace l'ensemble de l'instruction source dans la destination

Naturellement, les mêmes modificateurs peuvent être utilisés pour toutes les instructions, et pas seulement pour les `MOV` . Certaines instructions comme `JMP` et `SPL` , cependant, ne se soucient pas des modificateurs. (Pourquoi devraient-ils? Ils n'ont pas manipuler des données réelles, ils ont juste sauter partout.)

Comme tous les modificateurs de sens pour toutes les instructions, ils seront par défaut la plus proche qui fait sens. Le cas le plus courant implique l' `I.` modificateur: Pour garder la langue simple et abstrait aucun équivalents numériques ont été définies pour les OpCodes, donc en utilisant des opérations mathématiques sur les aurait pas de sens du tout. Cela signifie que pour toutes les instructions à l'exception `MOV` , `SEQ` et `END` (et `CMP` qui est juste un alias pour `SEQ`) le `je.` modificateur le même sens que l' `. F` .

Une autre chose à retenir sur l' `. Je` et le `. F` est que les modes d'adressage sont aussi partie de l'opcode, et ne sont pas copiés par `MOV . F`

Nous pouvons maintenant réécrire les anciens programmes à utiliser les modificateurs, par exemple. L'IMP serait naturellement `MOV . J'ai 0, 1` . Le Nain deviendrait:

```
        ADD . AB  # 4, 3
         MOV . J'ai   2, @ 2
         JMP      -2
         DAT      # 0, # 0
```

Notez que j'ai laissé de côté les modificateurs pour `JMP` et `DAT` , car ils ne les utilisent pas pour rien. Le MARS les transforme en (par exemple) `JMP . B` et `DAT . F` , mais qui s'en soucie?

Oh, encore une chose. Comment ai-je sais qui modificateur à ajouter à laquelle l'enseignement? (Et, plus important encore, comment le MARS ajouter eux si nous les laissons hors?) Eh bien, vous pouvez généralement le faire avec un peu de bon sens, mais la norme ne définit 94 un ensemble de règles à cet effet.

`DAT` , `NOP`
: Toujours `. F` , mais elle est ignorée.

`MOV` , `SEQ` , `SNE` , `CMP`
: Si A est en mode immédiat, `. AB` ,
si en mode B est immédiate et A-mode n'est pas, `. B` ,
si aucun mode est immédiat, `. I` .

`ADD` , `SUB` , `MUL` , `DIV` , `MOD`
: Si un mode est immédiat, `. AB` ,
si en mode B est immédiate et A-mode n'est pas, `. B` ,
si aucun mode est immédiat, `. F` .

`SLT` , `LDP` , `STP`
: Si A est en mode immédiat, `. AB` ,
si elle n'est pas, (always!) `. B` .

`JMP` , `JMZ` , `JMN` , `DJN` , `SPL`
: Toujours `. B` (mais elle est ignorée pour `JMP` et `SPL`).

Plongée profonde dans la norme 94

Le # est plus qu'il n'y paraît ...

Le comportement défini du mode d'adressage immédiat (°) dans la norme 94 est tout à fait inhabituel. Alors que la norme est 100% compatible avec l'ancienne syntaxe, l'adressage immédiat a été défini de façon très intelligente et unique qui lui permet de servir logiquement avec toutes les instructions et les modificateurs, et en fait un outil très puissant.

En regardant les modificateurs, vous pourriez vous demander ce que `MOV . F # 7, 10` ferait. `. F` devrait déplacer les deux domaines, mais il n'y a qu'un seul numéro à la source? Serait-il déplacer 7 dans les deux champs de la destination?

Non, il ne serait pas définitivement. En fait, il se déplacerait 7 dans le champ A-de la destination, et *10* dans le domaine de la B-! Pourquoi?

La raison en est que, dans la syntaxe '94, la source (et de la destination) est *toujours* un entier instruction. Dans le cas d'adressage immédiat, c'est tout simplement toujours l'instruction en cours, (c'est à dire 0) quelle que soit la valeur réelle. Donc, `MOV . F # 7, 10` se déplace deux champs de la source (0) à la destination (10). Surprenant, n'est-ce pas?

Les mêmes œuvres même pour `MOV . j'ai` . Cette façon de définir adressage immédiat nous permet également utiliser des instructions qui, même sans modificateurs, n'aurait pas de sens dans la norme comme '88 `JMP # 1234` . Évidemment, vous ne pouvez pas sauter dans un certain nombre, mais vous pouvez sauter dans l'adresse de ce numéro, ou 0. Cette offre de nombreux avantages évidents, car non seulement nous pouvons stocker des données dans le champ A pour «libre», mais le code va survivre même si quelqu'un décrémente. Nous pouvons maintenant réécrire le code imp de décision plus tôt pour être un peu plus robuste:

```
        SPL     # 0, } 1
         MOV . J'ai   # 1234, 1
```

Il travaille toujours le même, mais maintenant, les A-domaines sont libres. Juste pour le plaisir, je vous ai laissé le `SPL` incrémenter le A-domaine de l'imp, de sorte que tous les lutins seront différents. Depuis `SPL` n'utilise pas son champ de B, que l'incrément est aussi "libre". Il travaille, croyez-moi - ou essayer vous-même!

Modulo mathématiques

Vous devez déjà savoir que les adresses dans le noyau s'enroulent autour, de sorte que l'instruction un coresize devant ou derrière l'instruction en cours se réfère à l'instruction en cours lui-même. Mais en fait, cet effet est beaucoup plus profond: tous les numéros dans la guerre de base sont converties dans la gamme 0 - *coresize* -1.

Pour ceux d'entre vous qui connaissent déjà sur la programmation et champ limité entier maths, disons simplement que tous les numéros dans la guerre de base sont considérés comme non signé, avec l'entier maximum étant *coresize* -1. Si cela n'a pas clarifier tout, lisez la suite ...

En effet, tous les nombres de guerre du Coeur sont divisées par la longueur de l'âme, ou *coresize* , et que le reste est maintenu. Vous pouvez essayer de penser à une calculatrice avec un écran de seulement 8 chiffres qui jette des chiffres du passé qui, de sorte que 100 * 12345678 (1234567800, bien sûr) ne s'affiche (et stockées) comme 34567800. De même, en un noyau d'instructions 8000, 7900 222 (8122) ne devient 122.

Qu'advient-il de nombres négatifs, alors? Ils sont normalisées aussi, en ajoutant *coresize* jusqu'à ce qu'ils deviennent positifs. Cela signifie que ce que j'ai écrit que -1 est actuellement stocké par le MARS comme *coresize* -1, ou dans une instruction de base 8000, comme cela est courant, 7999.

Bien sûr, cela ne fait aucune différence pour les adresses, qui s'enroulent autour de toute façon. En fait, il ne fait aucune différence pour les instructions mathématiques simples comme `ADD` ou `SUB` soit, puisque avec *coresize* = 8000, 6 7998 donne le même résultat de 4 (ou 8004) comme le fait 6-2.

Quel est le problème, alors? Eh bien, il ya quelques instructions où il fait une différence. Ces instructions que `DIV` , `MOD` et `SLT` traitent toujours des nombres comme non signé. Cela signifie que -2/2 n'est pas -1, mais (*coresize* -2) / 2 = (*coresize* / 2) -1 (ou pour *coresize* = 8000, 7998/2 = 3999, pas 7999). De même, `SLT` considère -2 (ou 7998) pour être *plus grande* que 0! En fait, 0 est le plus petit nombre possible de guerre de base, de sorte que tous les autres numéros sont considérés comme plus grande qu'elle.

Le 94 instructions standard par l'instruction

Ok, votre patience a été récompensée. Jusqu'à maintenant, je vous ai donné que quelques morceaux de l'information. Maintenant il est temps de tout lier ensemble en décrivant chaque instruction pour vous.

Bien sûr, je pourrais les ai répertoriés au début, quand je vous poussé le jeu d'instructions , et probablement t'aurais sauvé de beaucoup

de deviner. Mais j'ai eu - au moins à mon avis - une très bonne raison d'attendre. Non seulement je veux vous montrer un peu de code pratique avant d'entrer dans les questions théoriques ennuyeux, mais la plupart de tout ce que je voulais que vous saisissez au moins l'idée de base des modes et des modificateurs d'adressage avant de décrire les instructions en détail. Si j'avais décrit les instructions avant les modificateurs, j'aurais dû d'abord vous enseigner les '88 anciennes règles, et plus tard enseigner tout nouveau avec des modificateurs inclus. Ce n'est pas une mauvaise façon d'apprendre Redcode, mais il ferait ce guide inutilement compliqué.

`DAT`

À l'origine, comme son nom l'indique, `DAT` a été conçu pour stocker des données, tout comme dans la plupart des langues. Depuis la guerre en base vous souhaitez réduire le nombre d'instructions, etc stocker des pointeurs dans les parties non utilisées d'autres instructions est commun. Cela signifie que la chose la plus importante à propos de `DAT` est que l'exécution elle tue un processus. En fait, depuis le niveau 94 a pas d'instructions illégales, `DAT` est définie comme une instruction tout à fait légal, qui *supprime le processus en cours d'exécution de la file d'attente de processus* . On dirait que les cheveux en quatre, peut-être, mais de définir précisément l'évidence peut souvent sauver beaucoup de confusion. Les modificateurs n'ont aucun effet sur `DAT` , et en fait, certains Marses les supprimer. Cependant, n'oubliez pas que predecrementing et postincrementing sont toujours fait, même si la valeur n'est pas utilisée pour quoi que ce soit. Une chose inhabituelle sur `DAT` , une relique des normes précédentes, c'est que si elle a un seul argument, il est placé dans le *champ de B* .

`MOV`

`MOV` copie les données d'une instruction à l'autre. Si vous ne savez pas tout sur ce que déjà, vous devriez probablement relire les chapitres précédents. `MOV` est l'un des quelques instructions qui soutiennent `. j'ai` , et c'est son comportement par défaut si aucun modificateur est donnée (et si aucun des champs utilise adressage immédiat).

`AJOUTER`

`ADD` ajoute la valeur (s) de la source à la destination. Les modificateurs fonctionnent comme avec `MOV` , sauf que `. que je` n'est pas supporté mais se comporte comme `. F` . (Qu'est-ce qui `MOV . AB` + `DJN . F` -être?) Rappelez-vous aussi que tous les maths dans la guerre de base se fait [modulo coresize](#) .

`SOUS`

Cette instruction fonctionne exactement comme `ADD` , sauf pour une différence assez évident. En fait, toutes les instructions arithmétiques "-logique» fonctionnent à peu près la même chose ...

`MUL`

... Comme c'est le cas pour `MUL` trop. Si vous ne pouvez pas deviner ce qu'il fait, vous avez probablement raté quelque chose de très important.

`DIV`

`DIV` fonctionne aussi à peu près le même que `MUL` et les autres, mais il ya quelques choses à garder à l'esprit. Tout d'abord, c'est [la division non signé](#) , ce qui peut donner des résultats surprenants parfois. **Division par zéro tue le processus** , tout comme l'exécution d'un `DAT` , et laisse la destination restent inchangées. Si vous utilisez `DIV . F` ou `. X` pour diviser deux numéros à la fois et l'un des diviseurs est 0, l'autre division sera toujours fait comme d'habitude.

`MOD`

Tout ce que j'ai dit à propos de `DIV` s'applique ici aussi, y compris la division par zéro pièce. Rappelez-vous que le résultat d'un calcul comme `MOD . AB # 10, # -1` dépend de la taille du noyau. Pour le noyau 8000 instruction commune le résultat serait 9 (7999 mod 10).

`JMP`

`JMP` se déplace exécution à l'adresse de ses points A-terrain à. La différence évidente mais importante pour les instructions "mathématiques" est que `JMP` ne se soucie que de l'adresse, pas les données qui portent sur les points à. Une autre différence importante est que `JMP` n'utilise pas son champ de B pour rien (et donc ignore également son modificateur). Être capable de sauter (ou split) en deux adresses serait tout simplement trop puissant, et il ferait la mise en œuvre des trois prochaines instructions assez difficile. N'oubliez pas que vous pouvez toujours placer un incrément ou un décrément dans le champ B utilisé, avec de la chance d'endommager le code de votre adversaire.

`JMZ`

Cette instruction fonctionne comme `JMP` , mais au lieu d'ignorer son champ de B, il teste la valeur (s) il pointe et ne saute si c'est zéro. Sinon, l'exécution se poursuivra à l'adresse suivante. Comme il n'y a qu'une seule instruction à tester, le choix de modificateurs est assez limité. `. AB` signifie la même chose que `. B` , `. BA` le même que `. A` , et `. X` et `. J'ai` le même que `. F` . Si vous testez les deux champs d'une instruction avec `JMZ . F` , il se placera *seulement si les deux champs sont nuls* .

`JMN`

`JMN` fonctionne comme `JMZ` , mais saute si la valeur testée est *pas* zéro (surprise, surprise ...). `JMN . F` saute si *l'un des champs est non nul* .

`DJN`

`DJN` est comme `JMN` , mais la valeur (s) sont décrémenté d'une *avant* les essais. Cette instruction est utile pour faire un compteur de boucle, mais il peut également être utilisé pour endommager votre adversaire.

`SPL`

C'est le grand. L'ajout de `SPL` dans la langue était probablement le changement le plus important jamais fait à Redcode, n'a d'égal que peut-être par l'introduction de la norme ICWS 94. `SPL` fonctionne comme `JMP` mais l'exécution *aussi* continue à l'instruction

suivante, de sorte que le processus est "scinder" en deux nouveaux. Le processus à l'instruction suivante exécute *avant* celui qui a sauté à une nouvelle adresse, qui est une petite mais *très* importante détail. (Beaucoup, sinon la plupart, des guerriers modernes ne seraient pas travailler sans elle!) Si le max. nombre de processus a été atteint, `SPL` fonctionne comme `NOP` . Comme `JMP` , `SPL` ignore son champ de B et son modificateur.

`SEQ`

`SEQ` compare deux instructions, et saute l'instruction suivante si elles sont égales. (Il saute toujours que les deux instructions avant, car il n'y a pas de place pour une adresse de saut.) Depuis les instructions ne sont comparées que pour l'égalité, en utilisant le `je.` modificateur est pris en charge. Tout naturellement, avec les modificateurs `. F` , `. X` et `. je` l'instruction suivante sera ignoré que si *tous* les champs sont égaux.

`END`

Ok, vous l'aurez deviné. Cette instruction saute l'instruction suivante si les instructions qu'il compare ne sont pas égaux. Si vous comparez plus d'un domaine, l'instruction suivante est ignorée si *une* paire d'entre eux ne sont pas égaux. (Cela vous semble familier, n'est-ce pas? Tout comme avec `JMZ` et `JMN` ...)

`CMP`

`CMP` est un alias de `la SEQ` . Ce fut le seul nom de l'instruction avant `SEQ` et `SNE` ont été introduits. Aujourd'hui, il n'a pas vraiment d'importance avec le nom que vous utilisez, car les programmes les plus populaires MARS reconnaissent `SEQ` même en mode 88.

`SLT`

Comme les instructions précédentes, `SLT` saute l'instruction suivante, ce temps si la première valeur est inférieure à la seconde. Comme il s'agit d'une comparaison arithmétique au lieu d'une logique, il ne fait aucun sens d'utiliser `. j'ai` . Il pourrait sembler qu'il devrait y avoir une instruction appelée `SGT` , (*sauter si supérieur à*), mais dans la plupart des cas, le même effet peut être obtenu simplement en échangeant les opérandes de `SLT` . Rappelez-vous que toutes les valeurs sont considérées comme non signé , donc 0 est le plus petit nombre possible et *-1 est le plus grand* .

`NOP`

Eh bien, cette instruction ne fait rien. (Et il le fait très bien, aussi.) C'est presque jamais utilisé dans un guerrier réelle, mais il est très utile pour le débogage. Rappelez-vous que des diminutions dans tout ou sont encore évalués.

Vous remarquerez peut-être que deux instructions, à savoir `LDP` et `STP` sont manquants. Ils sont un ajout relativement récent à la langue, et seront discutés ... euh, bien en ce moment. :-)

P-espace - la dernière frontière

P-espace est la dernière addition à Redcode, introduit par pMARS 0,8. Le «P» signifie privé, permanent, personnel, pathétique et ainsi de suite, comme vous voudrez. Fondamentalement, l'espace de P est une zone de mémoire qui ne peut accéder à votre programme, et qui survit entre les rounds dans un match multi-tour.

L'espace-P est à bien des égards différentes du noyau normal. Tout d'abord, chaque emplacement P-espace ne peut stocker un nombre non entier d'une instruction. En outre, l'adressage P-espace est absolue, c'est à dire. l'adresse P-espace 1 est toujours 1, peu importe où dans le noyau contenant l'instruction, il est. Et last but not least, l'espace de P ne peut être accessible par deux instructions spéciales, `LDP` et `STP` .

La syntaxe de ces deux instructions est un peu inhabituel. Le `STP` , par exemple a une valeur ordinaire dans le noyau en tant que sa source, qui est mis dans le domaine P-espace pointé par la destination. Donc, l'emplacement P-espace n'est pas déterminée par la destination *adresse* , mais par sa *valeur* , c'est à dire. la valeur qui serait écrasé s'il s'agissait d'un `MOV` .

Alors `STP . AB # 4, # 5` , par exemple mettrait la *valeur* 4 dans le *champ P-espace* 5. De même,

```
        STP . B   2, 3
         ...
        DAT      # 0, # 10
         DAT      # 0, # 7
```

mettrait la valeur 10 dans le champ P-espace **7** , pas 3! Cela peut devenir assez déroutant si le `STP` lui-même utilise l'adressage indirect, qui mène à une sorte de système d'adressage "double-indirect".

`LDP` fonctionne de la même façon, sauf que maintenant la source est un champ P-espace et la destination d'une instruction de base. L'emplacement P-espace 0 est un endroit spécial en lecture seule. Toutes les écritures, seront ignorés, et il est initialisé à une valeur particulière avant chaque tour. Cette valeur est -1 pour le premier tour, 0 le programme est mort lors du tour précédent, et par ailleurs le nombre de programmes survivants. Cela signifie que, pour un-à-un matches, 0 signifie une perte, 1 victoire et 2 cravate.

La taille de l'espace-P est généralement plus petit que celui du coeur, typiquement 1/16 de la taille du noyau. Les adresses dans l'espace P enveloppent comme dans le noyau. La taille de l'espace de P doit naturellement être un facteur de la taille du noyau, ou quelque chose de bizarre qui se passera.

Il est un peu particularité dans la mise en œuvre de pMARS de P-espace. Depuis l'intention était de conserver l'accès à l'espace P-lente, le chargement ou la sauvegarde deux domaines P-espace avec une instruction n'est pas autorisée. C'est une bonne chose, mais le résultat est à tout le moins une bidouille. Ce que cela signifie réellement est que `LDP . F` , `. X` et `. j'ai` tout travail comme `LDP . B` ! (Et même pour `STP` aussi, bien sûr)

Absolument l'utilisation la plus courante de P-espace est de l'utiliser pour sélectionner une stratégie. Dans sa forme la plus simple, cela signifie sauver la stratégie précédente P-espace, et la commutation des stratégies si le champ P-espace 0 affiche le programme perdu la dernière fois. Ce genre de programmes sont appelés P-guerriers, P-commutateurs ou P-cerveau (prononcé *pois-cerveaux*).

Malheureusement, l'espace de P n'est pas aussi privé que ça. Tant que votre adversaire ne peut pas lire ou écrire votre P-espace directement vos processus peuvent être capturés et fait exécuter votre code adversaires, y compris `STP` s. Ce genre de technique est connue comme le lavage de cerveau, et tous les P-commutateurs doit être préparé pour cela, et pas paniquer si le champ de la stratégie contient quelque chose de bizarre.

L'analyseur

Les étiquettes et les adresses

Jusqu'à présent, j'ai écrit toutes les adresses dans nos exemples de programmes que les numéros d'instruction, par rapport à l'instruction en cours. Mais dans les grands programmes, cela peut devenir ennuyeux, pour ne pas mentionner difficile à lire. Heureusement, nous n'avons pas vraiment le faire, car Redcode nous permet d'utiliser des étiquettes, des constantes symboliques, des macros et toutes les autres choses que vous attendez d'un bon assembleur. Tout ce que nous devons faire est d'étiqueter les instructions une reportez-vous à eux avec les étiquettes, et l'analyseur calcule les adresses réelles pour nous, comme ceci:

```
imp:     mov . i     imp, imp 1
```

Whoa, ce qui s'est passé? C'est exactement le même programme que celui que je vous ai montré dans le début. Je viens remplacé le numérique adressée avec des références à un label, "imp". Bien sûr, dans ce cas de faire qui est assez futile. La seule instruction dans laquelle l'étiquette est utilisée est "IMP" lui-même, dans laquelle l'étiquette est remplacé par 0.

Avant de l'exécuter, l'analyseur dans le MARS convertit toutes ces étiquettes et d'autres symboles dans les nombres familiers. Un tel fichier de Redcode "pré-compilé" est appelé un *fichier de chargement* , pour une raison quelconque. Tous les Marses doivent être capables de lire des fichiers de chargement, mais certains peuvent ne pas avoir une vraie analyseur. Dans le format de fichier de chargement, le code précédent devient `MOV . J'ai 0, 1` . Nous aurions pu également écrit le même code que

```
imp:     mov . i     imp, à côté
prochaine:    dat       0, 0              , ou quel que soit
```

Dans ce cas, l'instruction étiquetée "next" est une instruction après "imp", il est donc remplacé par 1. Rappelez-vous que les adresses réelles sont toujours des nombres relatifs, si la Imp continueront d'être `MOV . J'ai 0, 1` , même après qu'il a lui-même copié vers l'avant sur "suivant".

En fait, l' `:` à la fin de l'étiquette n'est pas vraiment nécessaire. Je l'ai utilisé ici pour vous aider à voir où les étiquettes sont, mais je n'ai pas l'habitude j'utilise dans mes propres programmes. C'est une question de goût.

Oh, et juste au cas où vous vous demandez ce sujet, des instructions de RedCode sont insensibles à la casse. J'aime utiliser des minuscules pour les sources, car il semble plus agréable et majuscules uniquement pour le format compilé "de fichier de chargement" (la plupart du temps parce que c'est une tradition).

Le tout

Bien que les exemples des chapitres précédents peuvent compiler très bien, ils ne sont pas vraiment des programmes complets, mais des parties d'un. Un fichier de REDCODE contient des informations supplémentaires pour le MARS.

```
; REDCODE-94
; nommer  Imp
; auteur  A.K. Dewdney

        org      imp

imp:     mov . i     imp, imp une
         fin
```

Comme vous avez probablement déjà compris, tout un après `;` est un commentaire dans Redcode. Les lignes sur le dessus de ce programme, cependant, ne sont pas seulement des commentaires ordinaires. Le MARS les utilise pour obtenir des informations sur le programme.

La première ligne, `; REDCODE-94` , dit le MARS que c'est vraiment un fichier Redcode. Tout ce qui dépasse cette ligne est ignorée par le MARS. En fait, le MARS attend seulement une ligne *de départ* avec `, REDCODE` , mais on peut utiliser le reste de la ligne pour identifier la saveur de Redcode utilisé. Spécialement, les serveurs Koth lire cette ligne eux-mêmes, et l'utilisent pour identifier la colline du programme va.

L' `, le nom` et `, auteurs` lignes juste donner quelques informations sur le programme. Bien sûr, vous pouvez donner à n'importe quel format, mais en utilisant les codes spécifiques permet MARS lire les noms et les afficher lorsque le programme est exécuté.

La ligne avec le mot `FIN` - surprise, surprise - la fin du programme. Tout ce qui suit, il sera ignoré. Avec `; REDCODE` , il peut être utilisé par exemple pour inclure des programmes de RedCode dans l'e-mail.

La ligne avec `ORG` indique où l'exécution du programme doit démarrer. Cela nous permet de mettre d'autres instructions avant le début du programme. Le `ORG` commande est l'une des nouvelles choses incluses dans la norme 94. L'ancienne syntaxe, qui fonctionne toujours dans les programmes modernes aussi, est de donner l'adresse de départ comme argument à la `FIN` .

```
; REDCODE-94
; nommer  Imp
; auteur  A.K. Dewdney

imp:     mov . i    imp, imp 1

         fin      imp
```

Simple, compact, et malheureusement assez illogique. Et avec de longs programmes, vous devez faire défiler à la fin, juste pour voir où il commence. Dans la terminologie Redcode, tant `ORG` et `END` sont appelés *pseudo-OpCodes* . Ils ressemblent à des instructions réelles, mais elles ne sont pas effectivement compilés dans le programme.

Mais assez de la Imp. Voyons ce que le nain ressemblerait à Redcode moderne:

```
; REDCODE-94
, le nom  nain
; auteur  A.K. Dewdney
; stratégie  bombes cœur à intervalles réguliers.
; (légèrement modifié par Ilmari Karonen)
; affirmation CORESIZE% 4 == 0

         org      boucle

boucle:    ajouter . ab   # 4, bombe
          mov . i    bombe, @ bombe
          jmp      boucle
bombe:     dat      # 0, # 0

         fin
```

Les étiquettes font comprendre le programme beaucoup plus facile, n'est-ce pas? Remarquez que j'ai ajouté deux nouvelles lignes de commentaire. L' `; stratégie` en ligne décrit le programme brièvement. Il peut y avoir plusieurs de ces lignes dans le programme. La plupart des Marses actuels ignorent, alors vous pourriez aussi bien utiliser les commentaires ordinaires comme celui que mon nom est en, mais les collines afficher les `; stratégie` lignes à d'autres. Envoi du programme précédent à un, quelque chose comme cela pourrait être démontré:

```
Un nouveau challenger est apparu sur la colline '94!

Nain par AK Dewdney: (longueur 4)
; Bombes de la stratégie du noyau à intervalles réguliers.

[Autres informations ici ...]
```

L'environnement et, affirment

Un autre nouveau détail dans notre exemple de code est l' `; assert` ligne. Il peut être utilisé pour s'assurer que le programme fonctionne vraiment avec les paramètres actuels. Le Nain, par exemple, lui-même détruit si la taille de l'âme n'est pas divisible par quatre. Alors, j'ai utilisé la ligne `; affirmer CORESIZE% 4 == 0` pour s'assurer qu'il est toujours.

Le *CORESIZE* est une constante prédéfinie qui nous indique la taille du noyau. Autrement dit, *n* + *CORESIZE* est toujours la même adresse que *n* . Le `%` est l'opérateur modulo, qui donne le reste dans une division. La syntaxe des expressions utilisées dans le `; affirmer` lignes et ailleurs dans Redcode est le *même que dans le langage C* , bien que l'ensemble des opérateurs est beaucoup plus limitée.

Pour ceux qui ne connaissent pas C, voici une sorte de liste des opérateurs qui sont utilisés dans les expressions de RedCode:

Arithmétique:
`+` plus `-` soustraction (ou la négation) `*` multiplication `/` division `%` module (le reste)

Comparaison:
`==` égal `!=` ne pas égal `<` est inférieur à `>` supérieur à `<=` est inférieur ou égal à `>=` est égal ou supérieur à

Logique:
`&&` et `||` ou `!` pas

Affectation:
`=` affectation à une variable de

L' `; assertion` est suivi par une expression logique. Si elle est fausse, le programme ne sera pas compilé. En C, une valeur de 0 signifie faux et rien d'autre signifie vrai. Les opérateurs logiques et de comparaison retournent 1 pour vrai, un fait qui peut être utile plus tard.

Typiquement, `; assert` est utilisé pour vérifier que la taille du noyau est une des constantes ont été conçus pour, comme `; affirmer CORESIZE == 8000` . Si le programme utilise P-espace, son existence peut être testé avec `; affirmer PSPACESIZE> 0` . Depuis notre exemple, le nain, est assez souple, je n'ai testé que la *CORESIZE* pour divisibilité, pas pour une taille spécifique. Le diablotin, qui fonctionne avec *des* paramètres, pourrait utiliser `; valoir 1` , `; valoir 0 == 0` et ainsi de suite, qui toujours évaluer aussi vrai. Ceci est utile car sinon le MARS peut se plaindre d'une "`manquantes; affirmer ligne - guerrier peut ne pas fonctionner avec les paramètres actuels.`"

Certains des constantes prédéfinies, tels que *CORESIZE* , sont définis par la norme 94, et d'autres peuvent et ont été ajoutés. pMARS 0,8 devrait supporter au moins les éléments suivants:

- *CORESIZE* - la taille de la base (par défaut 8000)
- *PSPACESIZE* - la taille de l'espace de P (par défaut 500)
- *MAXCYCLES* - le nombre de cycles jusqu'à une cravate est déclarée (par défaut 80000)
- *MaxProcesses* - la taille maximale de la file d'attente de processus (par défaut 8000)
- *WARRIORS* - le nombre de programmes dans le noyau (généralement 2)
- *MAXLENGTH* - la longueur maximale d'un programme (par défaut 100)
- *CURLINE* - le nombre d'instructions compilées à ce jour (0 à *MAXLENGTH*)
- *MINDISTANCE* - la distance minimale entre deux guerriers (par défaut 100)
- *VERSION* - la version de pMARS, multiplié par 100 (80 ou plus)

Define? Enfin, presque ...

Les constantes prédéfinies sont utiles, et sont donc des étiquettes, mais est-ce vraiment tout? Puis-je utiliser des variables ou quelque chose?

Eh bien, Redcode est un langage d'assemblage, et ils ne utilise pas beaucoup de variables. Mais il ya quelque chose de presque aussi bon, ou peut-être parfois même mieux. Il ya une pseudo-OpCode `EQU` qui nous permet de définir nos propres constantes, des expressions et même les macros. Il ressemble à ceci:

```
étape     équ       2667
```

Après cela, *l'étape* est toujours remplacé par 2667. Il ya un hic, cependant. Le remplacement est textuel, pas numérique. Dans ce cas, il ne devrait pas faire de mal, mais tout cela fait `EQU` un outil très puissant, il peut créer des problèmes qui les programmeurs C doivent être assez familier avec. Prenons un exemple.

```
étape     équ       2667
```

```
cible   équ      étape-100

commencer    mov . i    cible, étape-cible
```

Le A-champ de la MOV serait 2567, tout comme il devrait être. Mais le champ de B deviendrait 2667-2667-100 == -100, pas 2667 - (2667-100) == 2667-2567 == 100, tel qu'il a été probablement destiné. La solution est simple. Il suffit de mettre des parenthèses autour de chaque expression dans EQU s, comme " `cible équ (étape 100)` ".

Avec les versions modernes de pMARS il est possible d'utiliser plusieurs lignes équ s, et ainsi créer une sorte de macros. La façon dont il l'a fait est la suivante:

```
Dec7     équ      dat  # 1, # 1
         équ      dat  $ 1, $ 1
         équ      dat  @ 1, @ 1
         équ      dat  * 1, * 1
         équ      dat  { 1, { 1
         équ      dat  } 1, } 1
         équ      dat  < 1, < 1

leurre Dec7
        Dec7             ; 21 instruction leurre
        Dec7
```

Qu'est-ce que c'est "rof" utilisé?

Il ya un peu plus de fonctionnalités de l'analyseur pMARS gauche, et celui-ci est peut-être plus puissant (et plus difficile à apprendre) que tout ce qui précède. Les POUR / ROF pseudo-OpCodes peuvent non seulement faire vos sources courte et créer facilement des séquences de codes complexes, mais ils peuvent être utilisés pour créer du code conditionnel pour des réglages différents.

Une POUR bloc commence avec - oui, vous l'aurez deviné - la pseudo-OpCode POUR , suivi par le nombre de fois que le bloc doit être répété. S'il ya une étiquette avant le bloc, il sera utilisé comme un compteur de boucle, comme ceci:

```
indice    pour      7
         dat      indice, 10 index
         rof
```

Le bloc se termine, comme vous pouvez le voir, avec ROF . (Bien mieux que le vieux cliché SUIVANT ou REPEAT ., je dirais) Le bloc ci-dessus serait analysé par pMARS dans:

```
        DAT . F    $ 1, $ 9
         DAT . F    $ 2, $ 8
         DAT . F    $ 3, $ 7
         DAT . F    $ 4, $ 6
         DAT . F    $ 5, $ 5
         DAT . F    $ 6, $ 4
         DAT . F    $ 7, $ 3
```

Il est tout à fait possible d'avoir plusieurs POUR blocs à l'intérieur de l'autre. Les blocs peuvent même contenir EQU s en leur sein, ce qui nous permet de créer un code très intéressant. Une caractéristique encore plus utile, c'est que le compteur de boucle peut être jointe à un marqueur avec la & -opérateur. Ceci est le plus couramment utilisé pour éviter le double étiquettes déclarant, mais il peut être utile pour d'autres fins.

```
dest01   équ      1000
dest02   équ      1234
dest03   équ      1666
dest04   équ      (CORESIZE-1111)

JTable
ix       pour      4
sauter et ix spl      dest & ix
         djn . b    saut et ix, # ix
         rof
```

Ce serait, après la DE / ROF est analysée, devenu:

```
JTable
jump01   spl      dest01
         djn . b    jump01, # 1
```

```
jump02   spl      dest02
         djn . b    jump02, # 2
jump03   spl      dest03
         djn . b    jump03, # 3
jump04   spl      dest04
         djn . b    jump04, # 4
```

Quant à ce que ce serait utile pour, eh bien, c'est à votre propre imagination. Les seuls guerriers que j'ai vu l'utilisation de ces expressions complexes sont quelques quickscanners. Les constantes prédéfinies peuvent également être utilisés avec `POUR` / `ROF` , comme ceci:

```
; Le corps de guerrier principal est ici

appeau
foo      pour      (MAXLENGTH-CURLINE)
         dat      1, 1
         rof

        fin
```

Cela remplit l'espace restant dans votre guerrier avec `DAT 1, 1` . Un tel leurre peut détourner les attaques des autres guerriers, à condition que vous avez copié (*démarré*) votre propre programme à partir du leurre. Notez que j'ai utilisé *foo* comme un compteur de boucle, même si il n'est pas utilisé pour quoi que ce soit. C'est parce que sinon le MARS considérerait *leurre* d'être un compteur de boucle à la place de l'étiquette, il devrait être.

Enfin, voici un exemple de certaines façons plus créatives d'utiliser `POUR` / `ROF` :

```
; REDCODE-94
; nommer  Tricky
, auteur  Ilmari Karonen
; stratégie  Certains truc de guerrier très complexe
; stratégie  (Un exemple d'auto-explicatif du code conditionnel)
; affirmer CORESIZE == 8000 | | CORESIZE == 800
 ; affirmer MaxProcesses> = 256 && MaxProcesses <10000
 ; affirmer MAXLENGTH> = 100

        org      début

        pour      0
 Il s'agit d'un bloc de commentaire pour / ROF. Ce sera répété 0 fois, qui
signifie que tout ici sera ignoré par le MARS. Il s'agit d'un
endroit idéal pour expliquer la stratégie complexe utilise ce guerrier.
        rof

; Bien sûr, l'utilisation des commentaires ordinaires est également possible. Vous pouvez utiliser
; selon alternative que vous aimez.

        pour      (CORESIZE == 8000)
étape     équ      normalstep
 ; Depuis une vraie comparaison renvoie 1 et un faux 0, ce morceau de
; code est compilé uniquement si la comparaison est vraie.
        rof

        pour      (CORESIZE == 800)
étape     équ      tinystep
 ;. Et ici nous pouvons mettre constantes optimisées pour la taille de base plus petite
        rof

        pour      0
 ; stratégie  Depuis stratégie et faire valoir les lignes sont des commentaires vraiment, ils
; stratégie  sera analysé à l'intérieur même des blocs 0 / ROF!
        ROF

; [Code réel ici ..]
```

Variété des variables

Le problème avec les constantes définies avec `EQU` , c'est qu'ils sont, ainsi, des constantes. Une fois que vous les avez défini, vous ne pouvez pas changer leurs valeurs. C'est très bien pour la plupart des fins, mais il fait quelques trucs sacrément près impossible.

Heureusement pMARS fournit quelques variables réelles pour que nous utilisions. Leur utilisation est un peu difficile et ça fait longtemps que je n'ai vu personne vraiment les utiliser, mais ils existent.

Les noms de variables ont une seule lettre, limitant ainsi leur nombre à 26 (*un* par *z*). Au lieu d'utiliser `EQU` , les variables sont assignées leurs valeurs avec le = opérateur. Le plus délicat est que, pour l'opérateur, on doit avoir une expression. Et puisque pMARS ne reconnaît pas l'opérateur virgule, il peut être nécessaire d'écrire des expressions factices.

Pourtant, les variables peuvent être utiles. Par exemple, la séquence suivante d'auto-généré Fibonacci serait probablement impossible sans eux.

```
        dat      # 1, (g = 0) + (f = 1)
idx      pour      15
         dat      # idx +1, (f = f + g) + ((g = fg) && 0)
         rof
```

Notez comment l'expression `(g = fg)` est «caché» par AND avec 0. Le système fonctionne parce pMARS ne réordonner l'expression mais toujours évaluée du côté gauche de la première addition, de sorte que lorsque le côté droit est en cours de calcul, *f* a déjà été augmentée.

Épingles et aiguilles

Bon, j'ai presque oublié. Il ya encore une pseudo-Op gauche à décrire. C'est presque jamais utilisé, mais oui, il est là. Le `code PIN` est synonyme de "P-espace numéro d'identification". Si deux programmes ont le même nombre que leur `NIP` , ils vous feront partager leur P-espace. Cela peut être utilisé pour fournir une sorte de communication inter-processus et même de la coopération. Malheureusement, la stratégie ne semble pas être la peine qu'il faut pour créer une méthode affective et rapide de communication. Bien sûr, si vous voulez l'essayer, allez-y. Vous ne savez jamais si ça va être un succès ...

Si le programme n'a pas de `code PIN` , leur P-espace sera toujours privé. Même si les deux programmes ne partagent leur P-espace, l'emplacement en lecture seule spéciale 0 est toujours privé.

Escalade de la colline

Si vous ne connaissez pas déjà à leur sujet, le *roi de la colline* serveurs (souvent appelé seulement *collines*) sont des tournois continus de guerre de base sur Internet. Guerriers sont envoyés par e-mail - ou inscrits sur un formulaire web - sur le serveur, qui oppose les agains tous les (habituellement 10-30) programmes déjà sur la colline. Le programme avec le plus bas score total tombe de la colline, et le nouveau guerrier remplacer (en supposant qu'il a obtenu un meilleur score que au moins un des programmes originaux). Il ya aussi un certain nombre de variations de cette configuration de base autour, comme des collines "infini", les collines de la diversité, etc

Notez que les collines généralement pré-compiler les guerriers dans des fichiers de charge avant de les courir pour gagner du temps. Cela peut conduire à certaines des constantes prédéfinies, telles que *WARRIORS* , étant incorrecte, et donc de mystérieux `; affirmer` problèmes.

Il ya actuellement (Avril 2012) deux serveurs principaux Koth disponible:

KotH.org
: Le plus ancien et le plus célèbre serveur Koth actif. Accueille actuellement 7 collines avec des paramètres différents, y compris les deux collines de mêlée multiwarrior et deux collines à l'aide de la plus Redcode 88 standard.

KOTH @ SAL
: Accueille également sept collines avec des paramètres différents, y compris un de la colline de débutants où les guerriers sont automatiquement poussés hors après avoir survécu 50 défis pour le rendre plus facile pour les nouveaux joueurs à le faire sur la colline.

En outre, la Koenigstuhl serveur héberge 10 collines "infinite" pour les guerriers publiés. Guerriers envoyés à ces collines ne sont jamais poussés hors, de sorte que les collines ne cessent de grossir. Le Koenigstuhl utilise également un argorithm de notation récursive qui règle la contribution d'un guerrier aux scores en fonction de son classement.

La liste ci-dessus n'est pas destinée à être exhaustive et est susceptible de devenir obsolète. Une liste plus détaillée et à jour des serveurs Koth actifs peut (a) se trouve à la page de corewar.info .

Histoire

v 0.50
: Fini le chapitre sur l'analyseur. (25 Mars, 1997)

v 0.51
: Correction d'un bug dans les exemples pour / rof

v 0.52
: *La première version publiée*

v 0.53
: Correction de quelques fautes de frappe et d'orthographe

v 0.54
: Ajout des 88 - les règles de conversion> '94

v 0.55
: Nettoyage du HTML un peu

v 1.00
: Ajout d'infos sur le = opérateur. Pourrait tout aussi bien appeler cette chose «version 1». (Le 5 mai 1997)

v 1.01
: Correction d'un mineur incompatibilité avec `<DD>` tags.

v 1.02
: Correction de quelques fautes de frappe et les phrases illogiques. Changement de la barre de navigation pour avoir un style commun avec le reste du site.

v 1.03
: Supprimé certaines images et aligner attributs, changé doctype Strict.

v 1.10
: *Aargh!* J'ai `SLT` arrière tout ce temps! Fixe. (8 Mars, 1998)

v 1.20
: Réécriture de beaucoup de escalade la colline afin de refléter la situation actuelle. Fait quelques autres changements mineurs dans le processus. Proposé le document à une nouvelle adresse à vyznev.net . Commuté à un Creative Commons licence. (Octobre 7, 2003)

v 1.21
: Ajouté couleurs (pour les navigateurs CSS permis)! Fait des changements et des corrections de typo plus mineur. Choisissez une orthographe standard et capitalisation pour le titre. C'est la première version publiée au vyznev.net . (11 Avril, 2004)

v 1.22
: Mise à jour de la licence de CC-By-NC 2.0 de CC-By 3.0, supprimer les restrictions sur l'utilisation commerciale. Suppression de la zone de notification de déplacement de page. Mise à jour la liste des serveurs Koth nouveau, enlever collines défunts. Aucun autre changement de contenu. (16 Avril 2012) Fréquence

---

Droit d'auteur