00: RandomJumpIfLessThan
Arguments: Value (1 byte), Address (1 word)
Generates a random number between 0 and 255. Then it compares it to a given value. If that number is strictly less than the given value, then it will jump to the given address.

01: RandomJumpIfMoreThan
Arguments: Value (1 byte), Address (1 word)
Same as above except if more than the given number, the script will jump to the address given.

02: RandomJumpOneIn256
Arguments: Address (1 word)
Generates a random number between 0 and 255. Has a one in 256 (about .4%) chance of jumping to the address given.

03: RandomJump255In256
Arguments: Address (1 word)
Same as above but has a 255 in 256 (about 99.6%) chance of jumping to the address given.

04: AddToViabilityScore
Arguments: Value (1 byte)
Adds the given amount to the viability score of the move currently being considered. 01, 02, 03… add 1, 2, 3… FF, FE, FD… subtract 1, 2, 3… If you put 0x5E, then you will automatically activate the switch move (make the AI switch if they can).

05: JumpIfHealthLessThan
Arguments: Target (1 byte), Value (1 byte), Address (1 word)
Looks as a given target (options: Attacker or Defender. All commands with targets will henceforth be presumed to only accept these two unless otherwise stated). Compares the percentage of their health remaining to a

given number. Jumps to the address given if health is strictly less than the percentage.

06: JumpIfHealthMoreThan
Arguments: Target (1 byte), Value (1 byte), Address (1 word)
Same as the command above except jumping if the value obtained is strictly more than the value given.

07: JumpIfHealthEquals
Arguments: Target (1 byte), Value (1 byte), Address (1 word)
Same as the command above except jumping if the value obtained is strictly equal to the value given.

08: JumpIfHealthNotEqual
Arguments: Target (1 byte), Value (1 byte), Address (1 word)
Same as the command above except jumping if the value obtained is strictly not equal to the value given.

09: JumpIfStatus1Equals
Arguments: Target (1 byte), Status Bits (1 word), Address (1 word)
Looks as a given target. Compares the bits in the RAM for status 1 to a certain amount, and if equal it will jump to the given address.

*It will jump if a bit matches. This means if you want to check for more than one condition at once, add the values of the conditions you want together. (For example, if you want to check for either sleep **or** poison, enter 00 00 00 0F).*

*This uses **or** logic, so if you enter in more than one condition at a time, the target only needs to have one for the command to return true.*

Values are stored in the game backwards, so if you want AA BB CC DD, the word needed is DD CC BB AA. Just use the value below as written and you will be good.

List of Status 1 Values:
00 00 00 07 —> Sleep (up to 7 turns)
00 00 00 08 —> Poison
00 00 00 10 —> Burn
00 00 00 20 —> Freeze
00 00 00 40 —> Paralyze
00 00 00 80 —> Badly poisoned

0A: JumpIfStatus1NotEqual
Arguments: Target (1 byte), Status Bits (1 word), Address (1 word)
*Same as above except it checks if the target does not have any of the statuses. The same principle for checking for more than one condition applies here too. I believe the same **or** logic applies here too, so if you check for more than one condition, the target only needs to not have one of them to jump.*

0B: JumpIfStatus2Equals
Arguments: Target (1 byte), Status Bits (1 word), Address (1 word)
Similar to Status 1, except with different conditions. Status 2 includes volatile conditions such as confusion or substitutes.

List of Status 2 Values:
00 00 00 07 —> Confusion
00 00 00 08 —> Flinch
00 00 00 70 —> Uproar
00 00 0F 00 —> Bide
00 00 10 00 —> Multi-turn attack
00 00 E0 00 —> Wrap
00 0F 00 00 —> Infatuation

00 10 00 00 —> CHR raised
00 20 00 00 —> Transformed
00 40 00 00 —> Recharge
00 80 00 00 —> Rage
01 00 00 00 —> Substitute
02 00 00 00 —> Destiny Bond
04 00 00 00 —> prevent escape
08 00 00 00 —> Nightmare
10 00 00 00 —> Cursed
20 00 00 00 —> Foresight
40 00 00 00 —> Defense Curl
80 00 00 00 —> Tormented

0C: JumpIfStatus2NotEqual
Arguments: Target (1 byte), Status Bits (1 word), Address (1 word)
Same as the previous command, except in the negative.

0D: JumpIfStatus3Equals
Arguments: Target (1 byte), Status Bits (1 word), Address (1 word)
Same as Status 1 and Status 2. Status 3 includes conditions such as Leech Seed, Ingrain, or the semi invulnerable turns of moves such as Fly.

List of Status 3 Values:
00 00 00 04 —> Leech Seed
00 00 00 10 —> Lock On
00 00 00 20 —> Perish Song
00 00 00 40 —> In the air
00 00 00 80 —> Underground
00 00 01 00 —> Minimized
00 00 02 00 —> Charge
00 00 04 00 —> Ingrain
00 00 08 00 —> Yawn (start sleep)
00 00 10 00 —> Yawn (sleep next turn)

00 00 20 00 —> Imprison
00 00 40 00 —> Grudge
00 01 00 00 —> Mud Sport
00 02 00 00 —> Water Sport
00 04 00 00 —> Underwater

0E: JumpIfStatus3NotEqual
Arguments: Target (1 byte), Status Bits (1 word), Address (1 word)
Same as the above command but in the negative.

0F: JumpIfStatus4Equals
Arguments: Target (1 byte), Status Bits (1 word), Address (1 word)
Same as the previous Status commands, but with Status 4. Status 4 includes conditions such as having Light Screen or Reflect set up.

List of Status 4 Values:
00 00 00 01 —> Reflect
00 00 00 02 —> Light Screen
00 00 00 10 —> Spikes
00 00 00 20 —> Safeguard
00 00 01 00 —> Mist

10: JumpIfStatus4NotEqual
Arguments: Target (1 byte), Status Bits (1 word), Address (1 word)
Finally, the last of the Status commands. This one is the same as the previous one except in the negative.

11: JumpIfByteLessThan
Arguments: Value (1 byte), Address (1 word)
Compares the byte in the free variable to a given value, and jumps to the given address if the free variable is less than it.

12: JumpIfByteMoreThan

Arguments: Value (1 byte), Address (1 word)
Same as the previous command except jumping if the free variable is more than the given byte.

13: JumpIfByteEquals
Arguments: Value (1 byte), Address (1 word)
Same as the previous command, except jumping if the free variable and the given byte are equal.

14: JumpIfByteNotEqual
Arguments: Value (1 byte), Address (1 word)
Same as the previous command, except jumping if the free variable and the given byte are not equal.

15: JumpIfWordLessThan
Argument: Value (1 word), Address (1 word)
Compares the given word to the word in the free variable. If the free variable is less than the given word, it jump to the given address.

16: JumpIfWordMoreThan
Argument: *Value (1 word), Address (1 word)*
Same as the previous command except jumping if the free variable is more than the given word.

17: JumpIfWordEquals
Arguments: *Value (1 word), Address (1 word)*
Same as the previous command except jumping if the free variable and the given word are equal.

18: JumpIfWordNotEqual
Arguments: *Value (1 word), Address (1 word)*
Same as the previous command except jumping if the free variable and the given word are not equal.

19: JumpIfMoveIDEquals
Arguments: 1 halfword, *Address (1 word)*
If the move being considered while the script is running is equal to the halfword given, then it will jump to the given address.

1A: JumpIfMoveIDNotEqual
Arguments: 1 halfword, *Address (1 word)*
This is the negative of the previous command. As you've surely noticed by now, lots of these command are the negative versions of previous ones.

1B: JumpIfByteInList
Arguments: List Address (1 word), Jump Address (1 word)
Compares the free variable to every entry in the list at the given address. If the byte can be found in there, then the script will jump to the given address.

Useful for doing things such as comparing a target's ability to a list (after using the GetAbility command discussed later).

The list is formatted as a list of bytes in a row, terminated by 0xFF.

1C: JumpIfByteNotInList
Arguments: List Address (1 word), Jump Address (1 word)
Negative version of the previous command.

1D: JumpIfHalfwordInList
Arguments: List Address (1 word), Jump Address (1 word)
Compare the free variable to a list of halfwords. Jumps to the given address if any of those halfwords match the free variable. Useful for comparing to a list of moves.

The list is formatted as a list of halfwords (BB AA to represent AABB) terminated by 0xFFFF.

1E: JumpIfHalfwordNotInList
Arguments: List Address (1 word), Jump Address (1 word)
Negative of the previous command. Moving right along.

1F: JumpIfDamagingMoveInMoveset
Arguments: 1 word / 1 byte, 1 word
Checks the target's move set and jumps if there is a move that does damage. In default FireRed, this command only checks the attacker's moves. I rewrote this one because I like to make the AI a cheater who knows which moves you have. Some call it sadistic, but I like to call it making the game more fun. Anyway, all I did in the rewrite is add the ability to choose which target's move set you check.

20: JumpIfNoDamagingMoveInMoveset
Arguments: 1 word / 1 byte, 1 word
Same as the last command but in the negative.

21: GetBattleTurnCounter
Arguments: none
Gets the current turn of battle and stores it into the free variable. This is a total counter, so it counts the amount of turns since the battle started, not since a Pokemon entered the field (that is a different command that we'll find later).

22: GetType
Arguments: 1 byte
Gets the type of the Pokemon or move and stores it into the free variable. The arguments accepted as follows:

0 = Defender's primary Type

1 = Attacker's primary Type
2 = Defender's secondary Type
3 = Attacker's secondary Type
4 = Type of the considered move
5 = Type of the move in the free variable
6 = Attacker's partner's primary Type
7 = Defender's partner's primary Type
8 = Attacker's partner's secondary Type
9 = Defender's partner's secondary Type

23: GetPowerOfConsideredMove
Arguments: none
Gets the power of the move considered and stores the result into the free variable.

24: GetPowerOfStrongestMove
Arguments: none
Gets the power of the strongest move in the attacker's moves.

25: GetMoveLastUsed
Arguments: Target
Gets the move that was used most recently by the given target and puts it in the free variable.

26: JumpIfFreeVarEquals
Arguments: 1 byte, 1 word
Jumps to the given address if the free variable equals the given value. The only difference I can find between this and JumpIfByteEquals is that in JumpIfByteEquals the free variable is loaded before the AI Cursor (the AI Cursor gets the byte to compare). This appears to only be used with GetType.

27: JumpIfFreeVarNotEqual

Arguments: 1 byte, 1 word
Same as the last command except in the negative.

28: JumpIfMoveWouldHitFirst
Arguments: 1 byte, 1 word
If the target would out speed the opponent, then the script will jump to the given address.

29: JumpIfMoveWouldHitSecond
Arguments: 1 byte, 1 word
If the target is slower than the opponent, then the script will jump to the given address.

2A: GetPerishCount //New command! Not in default BPRE!//
Arguments: 1 byte
This is a new command that I wrote. It takes the target given and obtains the Perish Count (amount of turns left until Perish Song takes effect and the target faints). This can be used for having certain moves have priority on the turn before fainting or other such things.

Here's list of bytes returned:
32 = Perish Count 3
31 = Perish Count 2
30 = Perish Count 1 (turn before fainting)
00 = Not affected by Perish Song

Note: There are a few commands in here that just do nothing. I don't mean they have a pointless effect, i mean they literally do absolutely nothing. Their routines consist of "bx r14", which if you don't know, just skips the whole thing essentially.

Therefore they can be replaced with custom commands. I've written a few, which I'll include here but you can make your own and put them here too.

2B: GetSpikesLayer //New command! Not in default BPRE!//
Arguments: 1 byte
So the way the AI checks for Spikes in default FireRed is stupid. The way Spikes works is you can have up to three layers set up. Depending on how many layers are set up, a certain amount of damage is done to non-levitating foes when they switch in. The more layers, the higher the damage.

Well the way default FireRed is programmed it is impossible for the AI to know when to stop using the move, and also use the move to its full potential at the same time. This is because the game checks the RAM address that has the "Spikes exist on this side of the field" byte instead of the "This is how many layers of spikes there are" byte.

Thus I wrote this command so you can check for all three layers of Spikes before decreasing the move's priority. So now the AI will be able to set up Spikes properly.

2C: CountViablePokemonOnTeam
Arguments: 1 byte
Checks how many Pokemon that aren't fainted are left on the target's team and puts that number into the free variable.

2D: GetMoveID
Arguments: none
Gets the index number of the move being considered and and stores it into the free variable. Useful for use with the JumpIfByteInList command.

2E: GetMoveScriptID
Arguments: Move (1 Byte)
Gets the move script ID of either the move being considered (0) or the move/value in the free var (1), and puts that into the free var.

2F: GetAbility
Arguments: 1 byte
Gets the ability of the target and stores it into the free variable.

30: Callasm
Arguments: 1 word
The holy grail of all commands. Have a command written that isn't here? Don't go through all the trouble of repointing the table and crap, just use this! The possibilities with this are endless!

If you ASM command has arguments, you need to put them directly after the pointer to the routine (+1 of course because this is THUMB we're talking about). So if my routine took one argument (say the target), I would put something like

Callasm OFFSET+1
Defender

If you're using macros you need to put it on the next line for it to work.

31: JumpIfDamageBonusEquals
Arguments: Type/Move (1 byte), Value (1 byte), Address (1 word)
Gets the effectiveness of the move considered against the defender. The first argument is either the current move being considered, or you can tell if a Type would be super effective or not.

Then it compares the result to a value, and jumps if they are equal.

Values (for Types)
0 = Normal
1 = Fighting
2 = Flying

3 = Poison
4 = Ground
5 = Rock
6 = Bug
7 = Ghost
8 = Steel
9 = Fairy
A = Fire
B = Water
C = Grass
D = Electric
E = Psychic
F = Ice
10 = Dragon
11 = Dark
12 = Current Move

Values for effectiveness
0 = Immune
A = 1/4 effectiveness (A = 14 / 2 = 28 / 4, 10 = 20 / 2 = 40 / 4)
14 = 1/2 effectiveness (14 = 28 / 2, 20 = 40 / 2)
28 = Neutral (40)
50 = 2x effectiveness (50 = 28 * 2, 80 = 40 * 2)
A0 = 4x effectiveness (A0 = 50 * 2 = 28 * 4, 160 = 80 * 2 = 40 * 4)

32: JumpIfAnyOrAllStatsAre
Arguments, 4 bytes, 1 word
This command has a whopping five arguments! That sounds daunting, but the command itself is not too complicated.

The first argument is the target (Attacker or Defender).

The second argument is whether you want to check Any or All stats. All stats is every single stat for the target. Any just means if any of the target's stats fits the next argument, then the script will jump. Any is 0, All is 1.

The third argument is which comparison you want to make.

Values
0 = Equal
1 = NotEqual
2 = LessThan
3 = MoreThan

The fourth argument is the value you want to compare the stats too.

The final argument is the address you want to jump to.

33: DoesTeeterDanceWork
Arguments: none
This is a very specific command built to check whether or not Teeter Dance will have any effect. I wrote this because I didn't feel like adding two extra arguments to all eight Status commands to allow to check the partners in Double Battles. The only status inducing move I could think of that affected more than the attacker and defender was Teeter Dance.

This checks if any of the Pokemon are confused. If any aren't, it checks whether or not they have Own Tempo. It won't know if there is more than one Pokemon capable of being confused, but it will know if the move will have no effect at all.

This returns 1 if the move will work, and 0 if the move will fail.

34: JumpIfAnyPokemonHasStatus
Arguments: 1 byte, 2 words

Jumps to a given address if any party member on the target side has a given status. This checks for anything in the Status 1 category (talked about in the JumpIfStatus1Equals/NotEqual commands) since that is the only kind of status which persists after switching out.

35: JumpIfNoPokemonHasStatus
Arguments: 1 byte, 2 words
Exactly the same as the last command, but jumping if nobody in the party has the given status. This was apparently bugged in default FireRed, so I rewrote it (and also rewrote the previous command while I was at it).

36: GetWeather
Arguments: none
This command gets the current weather in the Battle Dimension and returns a value for it.

Values for Weather
0 = Clear
1 = Rain
2 = Sandstorm
3 = Sun
4 = Hail

37: JumpIfMoveScriptEquals
Arguments: 1 byte, 1 word
Jumps to the address given if the move script of the move being considered matches the given value.

38: JumpIfMoveScriptNotEqual
Arguments: 1 byte, 1 word
Negative of the previous command. Not much else to say here. Moving along.

39: JumpIfStatBuffLessThan
Arguments: 3 bytes, 1 word
This takes four arguments. The first is the target wanted, the second is the stat to compare, the third is the value to compare it to, and the final argument is the address to jump to if the value of the stat is strictly less than the value given.

Values for stats
1 = Attack
2 = Defense
3 = Speed
4 = Special Attack
5 = Special Defense
6 = Accuracy
7 = Evasion

Values for buffs
0 = -6
1 = -5
2 = -4
3 = -3
4 = -2
5 = -1
6 = No buffs
7 = +1
8 = +2
9 = +3
A = +4
B = +5
C = +6

3A: JumpIfStatBuffMoreThan
Arguments: 3 bytes, 1 word

Same as the last command except jumping if the stat is strictly more than the value.

3B: JumpIfStatBuffEqual
Arguments: 3 bytes, 1 word
Same as the last command, except jumping if the stat is equal to the given value.

3C: JumpIfStatBuffNotEqual
Arguments: 3 bytes, 1 word
Same as the last command except jumping if the stat is not equal to the given value.

3D: JumpIfMoveKnocksOut
Argument: 1 word
I believe this calculates the damage an attack will do damage equal to the opponent's HP before adding damage modification or critical-hit multipliers, and jumps if the attack will knock out the opponent.

3E: JumpIfMoveDoesntKnockOut
Arguments: 1 word
The same as the previous command except in the negative.

3F: JumpIfMoveInMoveSet
Arguments: 1, byte, 1 halfword, 1 word
The original I was not very fond of so I rewrote it. In the original, 1 checked the moves of the Attacker and 0 did something else involving both the attacker and defender.

In the new version, the target values are as follows:

0 = Defender
1 = Attacker

2 = Attacker's Partner
3 = Defender's Partner

This command also checks if the move is usable. It checks for Disable, Encore and Torment restrictions, as well as checking to see if the move is out of power points. If the move is there but not usable, then the command treats it as if it is not there.

40: JumpIfMoveNotInMoveSet
Arguments: 1 byte, 1 halfword, 1 word
The inverse of the command above. I have rewritten this one too using the same target values as the previous one.

41: JumpIfMoveScriptInMoveSet
Argument: 1 byte, 1 halfword, 1 word / 2 bytes, 1 word
I rewrote this one too. In the original, the command takes a halfword, which made little sense to me since move scripts only go up to FF (1 byte). So I just cut out that extra byte. Otherwise the rules are the same as in Command 3F with the same target values.

42: JumpIfMoveScriptNotInMoveSet
Arguments: 1 byte, 1 halfword, 1 word / 2 bytes, 1 word
Inverse of the previous command. This was also rewritten with the same syntax as before. Nothing much else to say.

43: JumpIfMoveSetRestricted
Arguments: 2 bytes, 1 word
Checks if the target is either Disabled (0) or Encored (1), and jumps to the given address if they are.

44: JumpIfEncoreIs
Arguments: 1 byte, 1 word

Checks if the active side is in Encore or not. To check if they are, use 0 as the argument. To check if they are not, use 1 instead. If the argument you picked matches the state of the active side, then the script will jump to the address given.

45: RunAway
Arguments: none
This makes the Pokemon flee on the next turn. This is used with the roaming Pokemon scripts. There's not terribly much use for it in Trainer Battles but it could be useful for a battle where the opponent leaves halfway through or when certain conditions are met (such as fleeing when health is under half).

46: GetMovePriority
Arguments: 1 byte
Gets the priority of the move in the given variable

47: SafariZone
Arguments: none
This triggers the Safari Zone text of "NAME is watching carefully." That's it.

48: GetHeldItemEffect
Arguments: 1 byte
Gets the held item effect byte of the given target and stores it into the free variable. Returns 0 if there is no item.

49: GetGender
Arguments: 1 byte
Gets the gender of the target and stores it into the free variable. Returns 0 for male, FE for female and FF for genderless.

4A: CheckIfFirstTurn
Arguments: 1 byte

Checks if it is the first turn that the target has been on the field. Returns 1 if yes, 0 if no. This is different from the total turns of battle, this is the individual number of turns. So when a Pokemon switches out, this is reset.

4B: GetStockpileCounter
Arguments: 1 byte
Gets the Stockpile counter for the given target and returns it into the free variable.

4C: CheckIfDoubleBattle
Arguments: none
Checks the battle type and sees if it is a double battle. If yes, then it returns 1, if not then it returns 0.

4D: **GetItemID**
Arguments: 1 byte
Gets the held item of the given target.

4E: GetKindOfMove
Arguments: 1 byte
I rewrote this command for the DPSS patch. Before this was seemingly identical to the GetType command. Now this checks the padding byte used by the DPSS patch to determine whether a move is physical, special or status.

Values used
0 = Physical
1 = Special
2 = Status

To apply the "rewrite" (it's not really a rewrite, it's byte change) change the bytes at 0xC945A from 80 78 to 80 7A.

4F: GetMovePriority
Arguments: Move/Var (1 byte)
Gets the priority of the considered move or free var move and puts it into the free variable.

50: **GetMoveRange**
Arguments: Move/Var (1 byte)
In default FireRed, this command got the move script ID of the move being considered or free var value move. So it did the exact same thing as the GetMoveScriptID command, thus rendering it utterly pointless.

I changed one little thing so that it was more useful. Now it gets the range of the move being considered. The range is how many Pokemon the move affects.

(Put this in the game by changing the bytes at 0xC94B2 to C0 79)

Most common values
0 = Single selected target
1 = Depends on the attack (i.e.moves like Counter target the last foe to hit them)
4 = Random target
8 = Both foes
10 = User
20 = Everyone but the user
40 = Opponent's field

51: GetProtectActivity
Arguments: 1 byte
Checks if the target has Endure or Protect (or any variation of those moves) active. Returns 1 if true, 0 if false.

52: GetTarget

Arguments: none
This command is really only useful in Double Battles where there is more than one option for your target. If it is a Single Battle, it returns 0 by default. If it is a Double Battle, it will return 0 or 2 if the Target is an opposing Pokémon (0 or 2 being the position of that Pokémon). If the target is the Partner, it will return 1.

53: GetSubHealthRatio
Arguments: 1 byte
Gets the health percentage of the Target's Substitute. Returns 0if there is no substitute up at the time.

54: CheckIfMoveFlagSet
Arguments: 2 bytes
Checks if the move flags for the considered move or free var value move have a bit in common with the byte given. If they do, then it returns 1.

Values:
0x1 —> Direct contact
0x2 —> Affected by Protect
0x4 —> Affected by Magic Coat
0x8 —> Affected by Snatch
0x10 —> Can be used by Mirror Move
0x20 —> Affected by King's Rock
0x3F —> All of the above

55: CheckIfInverseBattle
Arguments: None
This is to be used with Doesntknowhowtoplay's Inverse Battle Routine. It checks to see if flag 0x23D (the Inverse Battle flag) is set, and returns 1 if it is.

56: CheckIfStatsAre

Arguments: Target (1 byte), Stat 1 (byte), Stat 2 (1 byte), Comparison (1 byte)
Compares two stats for the given target. Returns 1 if the given comparison is true.

Values (Stats):
Attack = 1
Defense = 2
Speed = 3
Sp. Attack = 4
Sp. Defense = 5

Values (Comparison)
Equal = 0
NotEqual = 1
LessThan = 2
MoreThan = 3

57: GetDataAtRAM
Arguments: Size (1 byte), Address (1 word)
Gets the value at the given RAM address and puts it into the free variable. The size of the data gathered is the first argument. This can have a wide range of applications. For example, you can get the half word at 0x2023D68 to get the item lost from Knock Off.

Size Values
0 = Byte
1 = Half Word
2 = Word

58: Call
Arguments: 1 word

Calls another script at the address given and will return from where the previous script left off when finished. Returns with the ReturnToBattle command.

59: Jump
Arguments: 1 word
Jumps to another script at the given address. This does not return to the first script unless you jump back to it again.

5A: ReturnToBattle
Arguments: none
Stops the script and returns to normal battle. The game will then go to the next script in the AI Script table (or if this was the last one, then it will go to the next move in the attacker's move set and loop through the table again.) If this was the attacker's last move, then battle will resume (it resumes again at the start of the player's turn before they select an option).

5B: JumpIfBattlerLevelsAre
Arguments: 1 byte, 1 word
This command compares the attacker and the defender's levels to each other. I rewrote this one to have a few more options.

Original arguments
0 = Jump if the Attacker's level is higher than the Defender's
1 = Jump if the Attacker's level is higher than or equal to the Defender's
2 = Jump if the Attacker's level equals the Defender's

Rewrite's arguments
0 = Jump if the Attacker's Level equals the Defender's
1 = Jump if the Attacker's level does not equal the Defender's
2 = Jump if the Attacker's level is less than the Defender's
3 = Jump if the Attacker's level is more than the Defender's

5C: JumpIfTauntTurnsNotZero
Arguments: 1 word
Jumps to a given address if the amount of turns left in Taunt is not zero, (from my understanding this is when the defender has been Taunted in the first place). I said defender because this only checks the defender for some reason.

5D: JumpIfTauntTurnsZero
Arguments: 1 word
Same as the last command, except jumping when the amount of turns remaining in Taunt is zero (so this is probably when the foe has not been Taunted or is on the final turn of Taunt's activity).

Extra Commands With No Official Number (I've only used these with CallASM)

A Command To Get Accuracy